KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Toni Aaltola

PC- JA ANDROID-PELIEN OPTIMOIMINEN UNITYN ILMAISELLA VERSIOLLA

Opinnäytetyö
Marraskuu 2014

OPINNÄYTETYÖ
Marraskuu 2014
Tietojenkäsittelyn koulutusohjelma

Karjalankatu 3
80220 JOENSUU
p. 013 260 6800

Tekijä(t)
Toni Aaltola

Nimeke
PC- ja Android-pelien optimoiminen Unityn ilmaisella versiolla

Toimeksiantaja
-

Tiivistelmä
Tässä opinnäytetyössä käsitellään PC- ja Android-pelien optimoimista Unityn ilmaisella versiolla. Opinnäytetyö rajattiin siten, että sen fokus on opetella käyttämään Unityä ja siihen integroituja liitännäisiä sekä valmiita komponentteja mahdollisimman optimaalisesti hyödyksi. Koodien optimointia ei sen sijaan käsitellä kovinkaan syvällisesti.

Tämän lisäksi opinnäytetyöstä on rajattu pois ainoastaan Unityn Pro-versiossa toimivat optimointikeinot, jotta tätä opinnäytetyötä voisivat hyödyntää sekä Unityn ilmaisen että Pro-version käyttäjät. Myös alustakeskeiset optimointikeinot on rajattu pois, jotta opinnäytetyöstä olisi hyötyä kaikille käytetystä alustasta riippumatta.

Opinnäytetyö alkaa käytännön osiolla, jossa kerrotaan millaisia optimointikeinoja Unitylle on tarjolla. Tämän jälkeen optimointikeinoja testataan mahdollisuuksien mukaan kahdessa Unity-pelissä käyttäen testilaitteina PC-tietokonetta ja Android-täppäriä.

Testaamisen jälkeen tulokset arvioitiin, jotta saatiin selville mitkä optimointikeinoista olivat hyviä ja mitkä huonoja vastaamalla seuraaviin kysymyksiin: Kuinka tehokkaita optimointikeinot olivat peli- ja alustakohtaisesti? Vaikuttivatko optimointikeinot jotenkin pelien grafiikoiden tai muiden ominaisuuksien laatuun? Kuinka helppoa optimointikeinoja oli toteuttaa?

Kieli	Sivuja 101
suomi	Liitteet 0

Asiasanat
videopelit, optimointi, grafiikat, fysiikat

THESIS
November 2014
Degree Programme in Business Information Technology

Karjalankatu 3
80220 JOENSUU
FINLAND
Tel. +358 13 260 6800

Author (s)
Toni Aaltola

Title
Optimization of PC and Android Games in the Free Version of Unity

Commissioned by
-

Abstract
This thesis deals with optimization of PC and Android games in the free version of Unity. The thesis was restricted in such a way that its focus is in learning how to use Unity and its integrated plug-ins and ready-made components by the most optimal way. On the other hand, code optimizations are not dealt in depth.

In addition, the optimization methods that only work on Unity Pro version have been excluded, so that the thesis can equally advantage the users of Unity free and Pro version. Also, platform-specific optimization methods have been left out so that this thesis benefits all users, regardless of the platform they are using.

The thesis begins with the theoretical section, which describes what kind of optimization methods Unity features. After that the optimization methods were being tested, if possible, in two Unity games using PC computer and Android tablet as their testing devices.

After the testing the results were evaluated to find out which of the optimization methods were good and which were bad by answering the following questions: How effective optimization methods were on game-level and on platform-level? Did the optimization methods affect somehow to games' graphics or other characteristics of quality? How easy the optimization methods were to implement?

Language	Pages 101
Finnish	Appendices 0

Keywords
video games, optimizing, graphics, physics

Sisältö

1 Johdanto....13
2 Historiallisia optimointiesimerkkejä....14
2.1 Space Invaders....15
2.2 Pitfall!....17
2.3 Deus Ex....18
3 Pelien suunnitteluun liittyvät optimoinnit....20
4 Grafiikoiden optimointi....21
4.1 Prosessorin suorituskyvyn optimointi....21
4.1.1 Valaistuksen optimointi....23
4.1.2 Kameran asettelu....29
4.2 Grafiikkaprosessorin suorituskyvyn parantaminen....30
4.2.1 Tekstuurien optimointi....31
4.2.2 Kolmioiden määrän optimointi....33
5 Muut optimointikeinot....35
5.1 Tarpeettomien päivitysten välttäminen....35
5.2 Objektien altaus....37
5.3 Fysiikoiden optimointi 38
6 Optimoitavat Unity-pelit....40
6.1 Status: Insane....41
6.2 Puck Buddies....42
7 Testilaitteet....44
8 Optimointityökalut....46
8.1 Rendering Statistics Window....46
8.2 FPS Graph....48
9 Optimoimisen käytänteet....49
9.1 Alkutilanne ja testaamisen suorittaminen....50
9.2 Tietojen kerääminen....50
9.3 Laadun tarkkailu....52
9.4 Laatuasetusten säätäminen....53
10 Status: Insanen optimointi....56
10.1 Alkutilanne....57
10.2 Grafiikoiden suorituskyvyn pullonkaula....59
10.3 Valaistuksen optimointi....60
10.4 Kameran asettelu....64
10.5 Tekstuurien optimointi....66
10.6 Fysiikoiden optimointi....68
11 Puck Buddiesin optimointi....69
11.1 Alkutilanne....73
11.2 Grafiikoiden suorituskyvyn pullonkaula....74
11.3 Valaistuksen optimointi....75
11.4 Kameran asettelu....75
11.5 Tekstuurien optimointi....76
11.6 Fysiikoiden optimointi....78
12 Tulosten arviointi....80
12.1 Alkutilanne....81

12.1.1 Status: Insanen PC- ja Android-versiot....81
12.1.2 Status: Insanen PC-versio ja Puck Buddies....82
12.2 Grafiikoiden suorituskyvyn pullonkaula....84
12.3 Valaistuksen optimointi....84
12.3.1 Status: Insanen PC- ja Android-versiot....85
12.3.2 Status: Insanen PC-versio ja Puck Buddies....90
12.3.3 Yhteenveto....90
12.4 Kameran asettelu....90
12.4.1 Status: Insanen PC- ja Android-versiot....91
12.4.2 Status: Insanen PC-versio ja Puck Buddies....91
12.4.3 Yhteenveto....92
12.5 Tekstuurien optimointi....92
12.5.1 Status: Insanen PC- ja Android-versiot....92
12.5.2 Status: Insanen PC-versio ja Puck Buddies....93
12.5.3 Yhteenveto....94
12.6 Fysiikoiden optimointi....94
13 Loppusanat....95
Lähteet....98

Termit ja lyhenteet

3D-malli	3D-malli (engl. 3D model) on matemaattinen esitys kolmiulotteisesta objektista.
3D-mallin 2D-sovitus	3D-mallin 2D-sovituksessa (engl. UV mapping) 3D-malli levitetään yhdeksi tai useammaksi kaksiulotteiseksi pinnaksi.
Android	Android on käyttöjärjestelmä mobiililaitteille.
Anisotrooppinen tekstuuri	Anisotrooppinen tekstuuri (engl. anisotropic texture) on tekstuuri, jonka laatua parannetaan anisotrooppisella suodatuksella (engl. anisotropic filtering). Tällöin tekstuurit saadaan näyttämään terävämmiltä kaltevissa kulmissa ja kaukana kamerasta.
Buildi	Buildi (engl. build) on pelin versio, johon on liitetty kaikki tarpeelliset kirjastot ja muut tiedostot, jotta se toimii itsenäisesti ilman pelimoottoria.
Dynaaminen objekti	Dynaaminen objekti (engl. dynamic object) on objekti, joka liikkuu, kääntyy tai skaalautuu pelin aikana (usein esimerkiksi pelihahmot).
Dynaaminen valaistus	Dynaaminen valaistus (engl. dynamic lighting) on valaistus, joka lasketaan uudestaan jokaisen ruudunpäivityksen aikana.
Grafiikkaprosessori	Grafiikkaprosessori (engl. Graphic Processing Unit eli GPU) on tietokoneen osa, jonka tarkoitus on kiihdyttää ja suorittaa grafiikoiden piirtämistä.

Jäykkä kappale	Jäykkä kappale (engl. rigidbody) on fysiikan kappale, joka säilyttää muotonsa siihen kohdistuvista voimista huolimatta. Reaalimaailmassa mikään kappale ei voi olla täysin jäykkä, mutta videopeleissä jäykät kappaleet ovat varsin yleisiä, joissa niitä käytetään hyödyksi fysiikanmallinnuksessa.
Kolikkopeli	Kolikkopeli (engl. arcade game tai coin-op) on viihdelaite, jota käyttääkseen laitteeseen täytyy syöttää kolikkoja. Kolikkopelejä sijoitetaan tavallisesti julkisille paikoille, eikä niitä yleensä käytetä yksityiskäytössä.
Kolmio	Kolmiot (engl. triangle tai lyhennetysti tri) ovat kolmikulmaisia 3D-mallien pintoja, jotka määritellään janojen ja verteksien avulla.
Kovakoodaus	Kovakoodauksella (engl. hard coding, hard-coding tai hardcoding) tarkoitetaan tapaa ohjelmoida jokin toiminnallisuus siten, että toiminnallisuuden tarvitsee toimia oikein ainoastaan tilanteessa, johon se on alun perin ohjelmoitu sen sijaan, että toiminnallisuudesta ohjelmoitaisiin yleispätevä.
Kuhmujen kartoitus	Kuhmujen kartoituksella (engl. bump mapping) tarkoitetaan tekniikkaa, jolla 3D-mallin pinta saadaan näyttämään epätasaiselta ilman, että 3D-mallin verkon muotoon tehdään muutoksia.
Leivottu valaistus	Leivottu valaistus (engl. baked lighting) on valaistus, joka on muutettu yhdeksi tai useammaksi tekstuuritiedostoksi, jotka lisätään uusiksi kerroksiksi muiden tekstuurien päälle.

Liitännäinen	Liitännäinen (engl. plugin tai plug-in) on toiseen tietokoneohjelmaan liitettävä tietokoneohjelma, joka toimii vuorovaikutuksessa isäntäohjelmansa kanssa, eikä liitännäisten käyttäminen itsenäisinä ohjelmina ole yleensä edes mahdollista.
Painopisteiden koostaminen	Painopisteiden koostaminen (engl. blend weights) vaikuttaa siihen, kuinka monta luuta (engl. bone) voi vaikuttaa johonkin 3D-mallin yksittäiseen verteksiin maksimissaan animaatioiden aikana.
Mip-kartoitus	Mip-kartoitus (engl. mipmapping tai MIP mapping) on tekniikka, jossa jostain tekstuurista tehdään epätarkempia versioita, joita käytetään 3D-mallin pieniin kolmioihin. Tällöin ainoastaan suurimpiin kolmioihin käytetään tekstuurin alkuperäistä tarkkuutta.
Mobiililaite	Mobiililaite (engl. mobile device) on yleisnimitys erilaisille mukana kuljetettaville laitteille, joilla pääsee tietoverkkoon ajasta ja paikasta riippumatta. Mobiililaitteita ovat esimerkiksi älypuhelimet ja täppärit.
Muunnettu aika-askel	Muunnettu aika-askel (engl. fixed timestep) määrittelee Unityssä fysiikoiden ja FixedUpdate-funktioiden päivitysten määrän sekuntia kohden. Muunnetun aika-askeleen yksikkö on päivitysten välinen intervalli sekunteina: esim. 0.02 tarkoittaa sitä että päivitys suoritetaan 0.02 sekunnin välein.
Objektien altaus	Objektien altaus (engl. object pooling) on tekniikka, jossa objekteja kierrätetään sen sijaan, että luotaisiin jatkuvasti uusia ja tuhottaisiin vanhoja.

Partikkelisysteemi	Partikkelisysteemi (engl. particle system) on tekniikka, jolla voi simuloida esimerkiksi lumisateen, tulen tai hiuksien kaltaisia luonnollisia ilmiöitä käyttäen suurta määrää yksinkertaisia graafisia objekteja, kuten esimerkiksi spritejä.
PC	PC (lyhenne englannin kielen sanoista personal computer) on yleistietokone, jolla voi pelaamisen lisäksi tehdä monipuolisesti myös muuta.
Pelimoottori	Pelimoottori (engl. game engine) on pelien tekemiseen tarkoitettu ohjelmistokehys.
Pelin optimointi	Pelin optimointi (engl. optimizing game) on erilaisten toimenpiteiden tekemistä, jotta peli saadaan toimimaan mahdollisimman laadukkaasti, mahdollisimman vähäisillä laitteiston resursseilla. Pelien tehokkaassa optimoinnissa keskeistä on löytää suorituskyvyn pullonkaulat ja ratkaista ne.
Pelikonsoli	Pelikonsoli (engl. video game console) on erityisesti konsolipelien pelaamiseen tarkoitettu viihdelaite, joka kytketään tavanomaisesti televisioon.
Piirtokutsu	Piirtokutsu (engl. draw call) on grafiikkarajapinnalle, kuten esimerkiksi OpenGL tai Direct3D, lähetettävä kutsu piirtää jokin objekti.
Piirtämispolku	Piirtämispolku (engl. rendering path) on tekniikka, jolla valaistus ja varjot piirretään.
Pikselikohtainen valaistus	Pikselikohtainen valaistus (engl. per-pixel lighting) on dynaaminen valaistus, joka lasketaan erikseen jokaiselle ruudun pikselille.

Prosessori — Prosessori (engl. Central Processing Unit eli CPU) on tietokoneen osa, jonka tarkoitus on suorittaa tietokoneohjelman sisältämiä konekielisiä käskyjä.

Pullonkaula — Pullonkaula (engl. bottleneck) on osa-alue, joka estää kokonaisuuden toimimisen täydellä kapasiteetilla.

Resoluutio — Resoluutio (engl. resolution) on kuvan muodostavien pikselien määrä, joita on sekä vaaka- että pystysuunnassa. Resoluutio ilmoitetaan muodossa pikselien määrä vaakasuunnassa × pikselien määrä pystysuunnassa (esim. 1920 x 1080).

Reunojen pehmennys — Reunojen pehmennys (engl. anti aliasing tai anti-aliasing) on tekniikka, jolla 3D-mallien reunojen sahalaitaisuutta voidaan vähentää.

Roskien kerääjä — Roskien kerääjä (engl. garbage collector) poistaa automaattisesti muistista tiedot, joihin peli ei tule enää viittaamaan, jolloin muistitila vapautuu uudelleenkäytettäväksi. Useissa ohjelmointikielissä (esim. C ja C++) ei ole lainkaan roskien kerääjää, vaan muistitilan vapauttamisesta on huolehdittava itse.

Simulointi — Simulointi (engl. simulating) on todellisuuden jäljittelyä. Esimerkiksi tietokoneella tehtävässä tietokonesimuloinnissa todellisuutta jäljitellään siten, että reaalimaailman objektien vuorovaikutuksesta tehdään matemaattinen esitys.

Sprite — Sprite (engl. sprite) on kaksiulotteinen, pikseleistä koostuva kuva, jonka tausta on yleensä läpinäkyvä. Spritejä käytetään tavanomaisesti 2D-peleissä, mutta niitä voidaan käyttää myös 3D-peleissä, jos niihin tarvitaan litteitä 2D-hahmo-

ja tai muita 2D-kuvia.

Staattinen objekti — Staattinen objekti (engl. static object) on objekti, joka ei liiku, käänny tai skaalaudu missään vaiheessa peliä (usein esimerkiksi seinät).

Taso — Taso (engl. scene) on pelin osa, jossa pelin eri objekteja käytetään, minkä vuoksi jokaisessa pelissä on oltava ainakin yksi taso, jotta sitä pystyttäisiin pelaamaan. Taso voi olla esimerkiksi jokin pelin kenttä tai valikko.

Tekstuuri — Tekstuurien (engl. texture) avulla 3D-mallien kolmioihin saadaan väriä ja/tai kuviointeja.

Tiedon pakkaaminen — Tiedon pakkaaminen (engl. data compressing) on tekniikka, jonka avulla tiedostojen kokoa voidaan pienentää tiivistämällä tietoa, koska tieto yleensä sisältää vähemmän todellista informaatiota kuin sen kuvaamiseen on käytetty. Tiedon pakkaamisen voi tehdä joko häviöllisesti tai häviöttömästi riippuen siitä, tuleeko tiedoston todelliseen informaatioon muutoksia pakkaamisen yhteydessä.

Täppäri — Täppäri (engl. tablet) on mobiililaite, jossa on suunnilleen samat ominaisuudet kuin älypuhelimissa, mutta täppärien kosketusnäytöt ovat älypuhelinten näyttöjä suurempia. Täppäriä kutsutaan myös nimillä tabletti, tablet-tietokone, taulutietokone ja sormitietokone.

Törmäytin — Törmäytin (engl. collider) on objektin ympärille laitettava kehikko, jonka avulla tehdään objektien törmäyksien tunnistamiset (engl. collision detection).

Valokartoitus	Valokartoitus (engl. lightmapping) on tekniikka, jonka avulla valaistus pystytään leipomaan.
Varjostin	Varjostin (engl. shader) on skripti, jonka avulla tehdään grafiikoiden piirtäminen.
Verteksi	Verteksi (engl. vertex) on piste, jolla ilmaistaan geometristen kappaleiden risteyskohtia.
Verteksivalaistus	Verteksivalaistus (engl. vertex lighting) on dynaaminen valaistus, joka lasketaan ainoastaan 3D-mallien vertekseihin, joiden avulla valaistus saadaan interpoloitua 3D-mallien pinnoille.

1 Johdanto

Opinnäytetyöni aiheena on löytää tapoja optimoida PC- ja Android-pelejä Unityn (Unity Technologies 2014a) ilmaisella versiolla 4.5.4f1. Aiheen valintaan päädyin, kun huomasin, että useiden toiminnallisuuksien toteuttamiseen oli tarjolla erilaisia vaihtoehtoja. Tällöin tulin väkisinkin miettineeksi, millaiset ratkaisut olisivat kaikkein optimaalisimpia toteuttaa.

Toinen syy miksi päädyin valitsemaan aiheen on se, että uskoisin tästä opinnäytetyöstä olevan minulle paljon hyötyä tulevaisuudessa, koska Unity on todella suosittu pelimoottori tällä hetkellä suomalaisissa pelifirmoissa (Kuorikoski 2014, 246). Tämän lisäksi suomalaisista pelifirmoista suurin osa on tällä hetkellä keskittynyt mobiilipeleihin (Kuorikoski 2014, 246), joten optimoinnin osaaminen on todennäköisesti hyödyllistä tietotaitoa mobiililaitteiden resurssien niukkuuden vuoksi.

Optimoinnin tarve ei kuitenkaan rajoitu pelkästään mobiilipeleihin, sillä esimerkiksi juuri alkaneesta uudesta pelikonsolisukupolvesta ennustetaan tulevan suunnilleen yhtä pitkä kuin sen ennätyspitkästä edeltäjästä, joka kesti 8 vuotta pitkä. Joidenkin analyytikkojen mukaan kyseessä voi olla jopa viimeinen varsinainen pelikonsolisukupolvi (Lomas 2014). Käytännössä tämä tarkoittaa joka tapauksessa sitä, että laitteistot tulevat todennäköisesti pysymään muuttumattomina vielä vuosia eteenpäin, mutta peleiltä tullaan silti vaatimaan entistä näyttävämpää ja näyttävämpää visuaalista antia, joka voidaan saavuttaa optimoimalla.

Optimointi on kokonaisuutena varsin laaja kokonaisuus, joten joitain rajauksia on jouduttu tekemään: opinnäytetyön fokus on opetella käyttämään Unityä ja siihen integroituja liitännäisiä sekä valmiita komponentteja mahdollisimman optimaalisesti hyödyksi. Koodien optimointia ei sen sijaan käsitellä kovinkaan syvällisesti, koska jo pelkästä tekoälyn optimoimisesta saisi mahdollisesti oman opinnäytetyönsä.

Tämän lisäksi opinnäytetyöstä on rajattu pois ainoastaan Unityn Pro-versiossa toimivat optimointikeinot, jotta opinnäytetyötä voisivat hyödyntää tasapuolisesti sekä Unityn ilmaisen että Pro-version käyttäjät. Myös alustakeskeiset optimointikeinot on rajattu pois, jotta opinnäytetyöstä olisi hyötyä kaikille käytetystä alustasta[1] riippumatta.

Opinnäytetyö alkaa teoriaosiolla, jossa aluksi paneudutaan hieman optimoinnin historiaan ja kerrotaan kuinka optimointia on tehty joitain vuosia sitten, mitä havainnollistetaan kolmen tunnetun esimerkkipelin avulla. Samalla arvioidaan, mitä näistä vanhoista optimointikeinoista voidaan soveltaa myös nykyisin Unityssä. Tämän jälkeen syvennytään tarkemmin siihen, millaisia optimointikeinoja juuri Unitylle on tarjolla. Tämä osio on toteutettu suurimmaksi osaksi referoimalla Unityn dokumentaatioita, mutta muitakin lähteitä on käytetty, minkä lisäksi olen havainnollistanut useita optimointikeinoja omilla yksinkertaisilla esimerkeillä ja kuvilla.

Opinnäytetyön käytännön osiossa teoriaosion väittämiä sovellettiin kahdessa Unity-pelissä käyttäen testilaitteina PC-tietokonetta ja Android-täppäriä. Samalla kunkin optimointikeinon tehokkuutta mitattiin ja vaikutusta grafiikoiden sekä muiden pelien ominaisuuksien laatuun arvioitiin. Tämän lisäksi arvioitiin eri optimointikeinojen toteuttamisen helppoutta, jotta saatiin selville, millaisilla optimointikeinoilla saadaan aikaan mahdollisimman helposti, mahdollisimman hyviä lopputuloksia.

2 Historiallisia optimointiesimerkkejä

Optimointia on jouduttu käyttämään videopeleissä suunnilleen yhtä kauan kuin videopelejä on ollut olemassa. Tähän lukuun on koottu erilaisia historiallisia op-

---

1 Unityn tukemat alustat ovat tällä hetkellä: iOS, Android, Windows Phone 8, Windows, Windows Store Apps, Mac, Linux, Web Player, PS3, PS4, PlayStation Vita, PlayStation Mobile, Xbox One, Xbox 360 ja Wii U (Unity Technologies 2014b).

timointiesimerkkejä kolmesta tunnetusta esimerkkipelistä, joista jokainen edustaa oman aikansa teknistä kärkeä (Waugh 2014; Honkala 2012a, 62–65; NowGamer 2011). Samalla arvioidaan, mitä näistä vanhoista optimointikeinoista voidaan soveltaa myös nykyisin Unityssä.

2.1 Space Invaders

Space Invaders (Taito Corporation 1978, kuva 1) on kolikkopeli, jossa pelaajan täytyy ampua maata lähestyviä avaruusaluksia. Laitteiston vähäisen RAM-muistin määrän takia peliä on optimoitu siten, että ruudulla pystyy olemaan ainoastaan yksi pelaajan ampuma ammus kerrallaan. Sen vuoksi pelaaja pystyy ampumaan uudestaan vasta, kun ammuttu ammus on joko tuhoutunut tai poistunut ruudulta. Kyseinen optimointikeino tekee pelistä melko luonnottoman, koska ammuksen elinaika voi vaihdella paljon. Ilman tätä optimointikeinoa pelin tekeminen olisi ollut käytännössä mahdotonta, minkä vuoksi sen käyttäminen on kuitenkin perusteltu. (Unity Manual 2012a.)

Kuva 1. Space Invaders -kolikkopeli (Calleja 2006).

Vastaavaa optimointiratkaisua, jossa objekteja kierrätetään sen sijaan, että luotaisiin jatkuvasti uusia ja tuhottaisiin vanhoja, kutsutaan objektien altaukseksi. Luvussa 5.2 kerrotaan tarkemmin, kuinka sitä voidaan soveltaa Unityssä.

2.2 Pitfall!

Pitfall! (Activision 1982, kuva 2) on Atari 2600 -pelikonsolille julkaistu tasohyppelypeli, jossa pelaaja seikkailee viidakossa keräten aarteita ja vältellen erilaisia vihollisia sekä muita vaaroja. Pitfall! oli aikanaan poikkeuksellisen laaja peli, sillä Atari 2600 -pelikonsoli oli suunniteltu yksinkertaisen Pong-pelin (Atari Inc. 1972) ehdoilla. (Honkala 2012a, 63.)

Kuva 2. Kuvakaappaus Pitfall!-videopelistä (Arnold 2012).

Pitfallin! grafiikat ovat yksinkertaista pikseligrafiikkaa, minkä vuoksi niitä on pystytty optimoimaan siten, että pelaaja-spriteä on monistettu ja väritetty uudelleen. Tämän jälkeen monistetuista spriteistä on tehty mm. puun oksat ja maassa vierivät tukit (Honkala 2012a, 62). Vastaavaa menetelmää ei kuitenkaan pystyisi hyödyntämään Unityssä, koska grafiikoiden tekeminen Unitylle ja Atari 2600:lle eroavat hyvin paljon toisistaan: Unitylle grafiikat piirretään tekstuuritiedostoista (Unity Manual 2014a), kun taas Atari 2600:lle pitää tehdä omat algoritminsä, jotka piirtävät grafiikat (Honkala 2012a, 62). Unityssä puun oksat ja muut yksityiskohdat voikin täten huoletta piirtää suoraan tekstuuritiedostoihin.

Pitfallia! on optimoitu myös siten, että joillekin objekteille, kuten maassa vieriville tukeille, on määritelty useampi sijainti sen sijaan, että jokainen tukki olisi oma

objektinsa (Honkala 2012a, 62). Myös Unityssä on usein hyötyä objektien yhdistämisestä. Siitä kerrotaan lisää luvussa 4.1.

Pitfallin! suurin yksittäinen haaste oli kuitenkin pelimaailma: pelimaailman jokaisesta ruudusta haluttiin erilainen, mutta keskusmuistia oli käytössä hyvin niukasti. Tämä tavoite saavutettiin optimointiratkaisulla, jota pelin tehnyt David Grane kutsuu polynomilaskuriksi. Polynomilaskuri luo satunnaisia lukujonoja, jotka ovat kuitenkin täysin ennustettavissa. Luodut lukujonot määrittelevät ruudun polkuelementin, esteet, puut ja maanalaisen tunnelin umpikujat. Polynomilaskuri pystyy palauttamaan 256 erilaista lukujonoa, mikä tarkoittaa sitä, että pelialue koostuu yhteensä 256:sta ruudusta. Polynomilaskuri vie tilaa kovalevyltä ainoastaan 50 tavua. (Honkala 2012a, 64–65.)

Polynomilaskurin kaltaisia suunnitteluratkaisuja ei tarvitse enää nykyisin käyttää optimointitarkoituksessa, koska nykyisin pelilaitteiden kovalevyillä on valtavan paljon enemmän keskusmuistia kuin Atari 2600 -pelikonsolissa[2]. Toki toisinaan voi olla tarpeen vähentää lopullisten pelien tiedostokokoa, mutta esimerkiksi Unityn manuaalissa (2014b) kerrotaan, että pullonkaulat ovat yleensä aivan muualla kuin skriptien koossa. Sen sijaan satunnaisen maaston generoimiseen polynomilaskurin kaltaisia ratkaisuja voidaan käyttää yhä edelleen, mistä hyvänä esimerkkinä on vastikään pelaamani Spelunky (Mossmouth 2009) -tasohyppelypeli: Spelunkyssä kentät luodaan satunnaisgeneraattorilla, jonka idea on samankaltainen kuin polynomilaskurin idea, mutta erilaisia muuttujia on paljon enemmän.

2.3 Deus Ex

Deus Ex (Eidos Interactive 2000, kuva 3) on tulevaisuuteen sijoittuva peli, joka yhdistelee elementtejä mm. rooli- ja räiskintäpeleistä. Deus Exissä pelaajalle

---

2 Atari 2600 -pelikonsolissa on ainoastaan 128 tavua keskusmuistia (Wikipedia 2014a), kun taas moderneissa mobiililaitteissa voi olla useita gigatavuja (Wikipedia 2014b) ja moderneissa PC-tietokoneissa jopa joitain teratavuja keskusmuistia (Wikipedia 2014c).

tarjotaan useanlaisia ratkaisuja edetä: suoran toiminnan lisäksi etenemään pääsee myös väkivallattomasti. Deus Exin valinnanvapaus oli aikoinaan poikkeuksellista, ja vastaavia suunnitteluratkaisuja on sovelluttu myöhemmin useissa 2000-luvun hiekkalaatikkopeleissä. (Honkala 2012b, 70–72.) NowGamerin artikkelissa (2011) kerrotaan, että esimerkiksi monet BioWaren kehittämät pelit, kuten Star Wars: Knights of the Old Republic (LucasArts 2003), Mass Effect (Microsoft Game Studios 2007) ja Dragon Age: Origins (Electronic Arts 2009) ovat ottaneet vaikutteita Deus Existä.

Kuva 3: Kuvakaappaus Deus Ex -videopelistä (Ready Up 2010).

Deus Exiä suunniteltiin puoli vuotta ennen kuin sen ohjelmoiminen aloitettiin. Puolessa vuodessa suunnitteludokumentteja oli kirjoitettu peräti 500 sivua, ja pelille oli asetettu poikkeuksellisen kunnianhimoiset tavoitteet. Se, kuinka realistisia tavoitteet olivat, testattiin tekemällä pelistä prototyyppi, joka sisälsi mm. autenttisen kokoisen Valkoisen talon. Prototyyppi paljasti, että Valkoinen talo kuormitti liikaa laitteiston resursseja, minkä vuoksi Valkoisen talon lisäksi useita muitakin peliin suunniteltuja suurimpia tasoja jouduttiin pilkkomaan pienemmiksi

osiksi. (Honkala 2012b, 71–72.) Tämän tyyppistä optimointia voidaan luonnollisesti soveltaa myös Unityssä, koska objektien yksinkertaistaminen ja vähentäminen laskee yleensä laitteiston resurssien kulutusta. Unityssä voi myös esimerkiksi kameran asettelulla vaikuttaa siihen, kuinka paljon eri objekteja joudutaan piirtämään kerralla. Tästä kerrotaan lisää luvussa 4.1.2.

Alun perin Deus Exin vaihtoehtoisten ratkaisumallien oli tarkoitus perustua simulointiin, mutta pelin prototyypin perusteella myös se oli käytännössä liian raskasta, minkä vuoksi jokainen vaihtoehtoinen ratkaisumalli jouduttiin kovakoodaamaan (Honkala 2012b, 72). Unityssä esimerkiksi simuloitua fysiikanmallinnusta tehdään jäykkien kappaleiden avulla, joissa on paljon erilaisia muuttujia, kuten esimerkiksi painovoima, massa ja kallistuskulma, jotta fysiikanmallinnuksesta saadaan realistisen näköistä (Unity Scripting API 2014a). Kovakoodatussa fysiikanmallinnuksessa sen sijaan näitä muuttujia ei tarvitse huomioida ollenkaan, minkä vuoksi se on yleensä simuloitua fysiikanmallinnusta kevyempää. Kovakoodattu fysiikanmallinnus ei kuitenkaan usein ole yhtä dynaamisen näköistä kuin simuloitu fysiikanmallinnus, mutta Unity tarjoaa myös muita keinoja optimoida fysiikoita, josta kerrotaan lisää luvussa 5.3.

3 Pelien suunnitteluun liittyvät optimoinnit

Pelien suunnittelussa tulee ottaa huomioon ensinnäkin kohdealusta: mikäli kohdealustassa on suuri määrä tehoja verrattuna pelin vaatimuksiin, optimoinnin tarve vähenee ja aika kannattanee tällöin käyttää olennaisempiin asioihin. Sen sijaan esimerkiksi näyttävän mobiilipelin tekeminen voi vaatia hyvinkin tarkkaa laitteiston resurssien käyttöä.

Unityn manuaalissa (2012b) on havainnollistettu pelien suunnitteluun liittyviä optimointeja Shadowgun-pelillä (Madfinger Games 2011) ja Sky Castle -tekniikkademolla (Stockton 2012), joista molemmat on tehty juurikin mobiililaitteita silmällä pitäen. Niistä käy ilmi, että jos käytetään esimerkiksi yhtä laskennallisesti

raskasta tehostetta, muista tehosteista voidaan joutua tinkimään: esimerkiksi Sky Castlessa käytetään kuhmujen kartoitusta, muttei lainkaan dynaamista valaistusta. Sen sijaan Shadowgunissa käytetään kuhmujen kartoitusta Sky Castlea rajoitetummin, mutta dynaaminen valaistus on käytössä, joskin hyvin rajoitetusti. Se, mihin tehosteisiin laitteiston resurssit kannattaa käyttää, riippuukin hyvin pitkälti pelistä.

4 Grafiikoiden optimointi

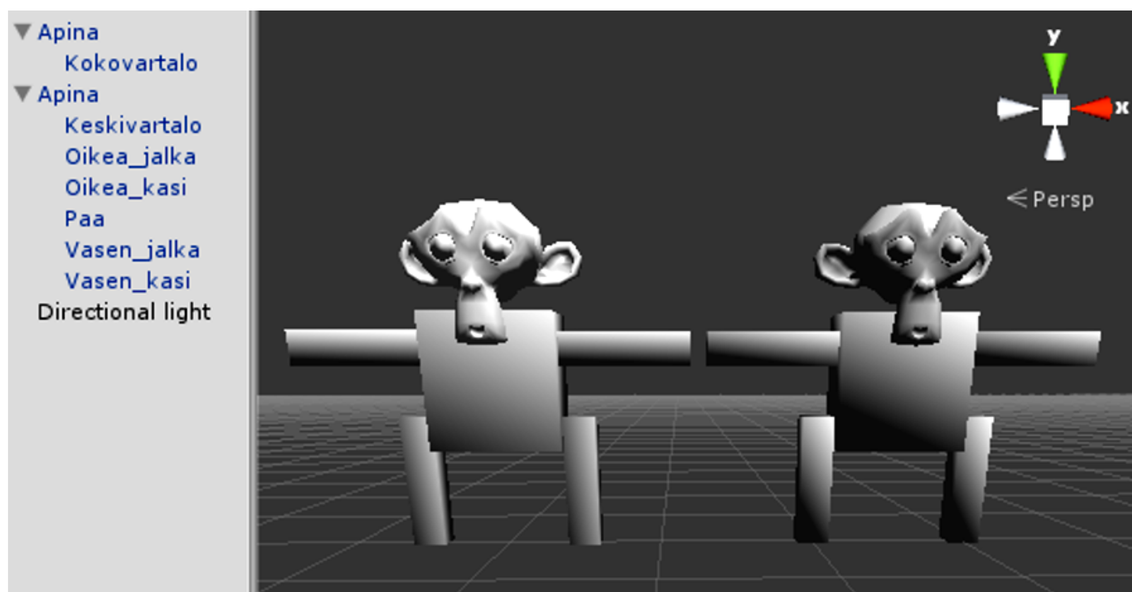
Ennen grafiikoiden optimoinnin aloitusta Unityn manuaalissa (2014c) kehotetaan selvittämään, onko suorituskyvyn pullonkaulana prosessori vai grafiikkaprosessori. Se on tärkeää, koska prosessorin kuormituksen optimointi voi lisätä grafiikkaprosessorin kuormitusta, kun taas grafiikkaprosessorin kuormituksen optimointi päinvastaisesti voi lisätä prosessorin kuormitusta. Tämän vuoksi väärään paikkaan tehdyllä optimoinnilla voi olla jopa suorituskykyä heikentävä vaikutus.

Pullonkaulan paikantamiseksi kehotetaan kokeilemaan, parantaako alhaisempi resoluutio suorituskykyä. Mikäli suorituskyky paranee, on pullonkaulana tällöin todennäköisesti grafiikkaprosessori. (Unity Manual 2014c.)

4.1 Prosessorin suorituskyvyn optimointi

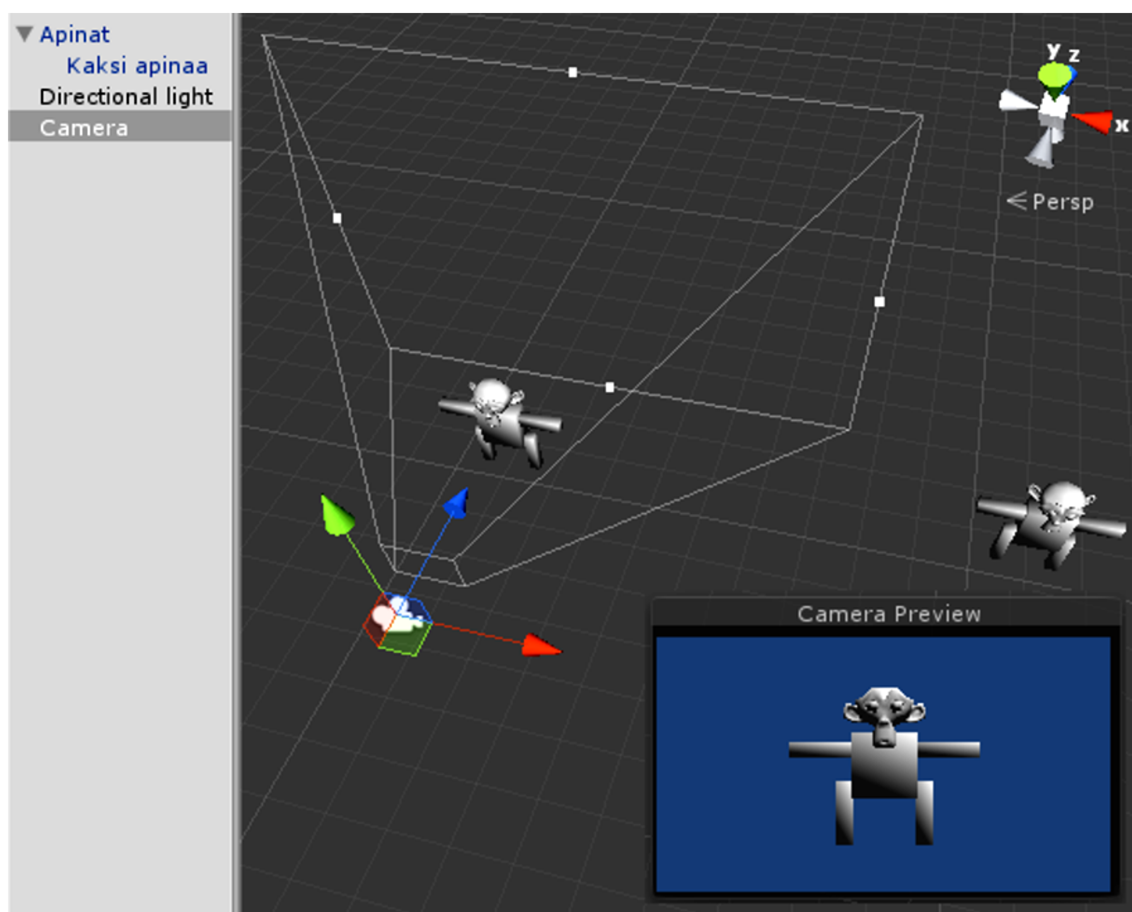
Prosessorin keskeisin tehtävä on käsitellä piirtokutsut. Piirtokutsut suoritetaan objektikohtaisesti, mikä tarkoittaa sitä, että yksi piirtokutsu vastaa yhtä piirrettyä (graafista) objektia. Tämän vuoksi toisiaan lähellä olevat objektit kannattaa usein yhdistää yhdeksi objektiksi, jotta prosessori voi käsitellä ne yhdellä piirtokutsulla useamman sijaan: esimerkiksi hahmomallin eri kehonosista ei yleensä kannata tehdä erillisiä objekteja, mikäli ei tarvitse (kuva 4). Grafiikkaprosessorin suorituskykyyn objektien yhdistäminen ei juurikaan vaikuta, koska objektien yh-

distämisellä ei ole vaikutusta 3D-mallien geometriaan. (Unity Manual 2014c.)



Kuva 4. Kuvakaappauksessa on kaksi muutoin identtistä apina-hahmoa, mutta toinen niistä koostuu yhdestä ja toinen kuudesta objektista.

Toisistaan kaukana olevia objekteja ei sen sijaan välttämättä kannata yhdistää, koska niistä saatetaan joutua piirtämään paljon sellaisia osia, jotka eivät näy kamerassa (kuva 5). Tämä ei lisää piirtokutsujen määrää, mutta se lisää tarpeetonta piirtämistä, mikä kuormittaa etenkin grafiikkaprosessoria. (Unity Manual 2014c.)



Kuva 5. Kuvakaappauksessa on yhdistetty yhdeksi objektiksi kaksi apinahahmoa, joista ainoastaan toinen näkyy kamerassa, kuten kameran esikatselusta (engl. camera preview) nähdään. Siitä huolimatta molemmat hahmoista joudutaan piirtämään piirtokutsujen objektikohtaisuudesta johtuen.

4.1.1 Valaistuksen optimointi

Unityssä on kolme erilaista piirtämispolkua: verteksivalaistu (engl. vertex lit), eteenpäin piirretty (engl. forward rendering) ja jaksotetusti valaistu (engl. deferred lighting) piirtämispolku. Piirtämispolun tyyppi vaikuttaa etenkin siihen, kuinka paljon pikselikohtainen valaistus kuormittaa laitteiston resursseja:

- Verteksivalaistussa piirtämispolussa ei voida käyttää pikselikohtaista valaistusta ollenkaan.
- Eteenpäin piirretyssä piirtämispolussa pikselikohtaisen valaistuksen kustannus on valaistujen pikselien määrä kerrottuna objektien määrällä.
- Jaksotetusti valaistussa piirtämispolussa pikselikohtaisen valaistuksen

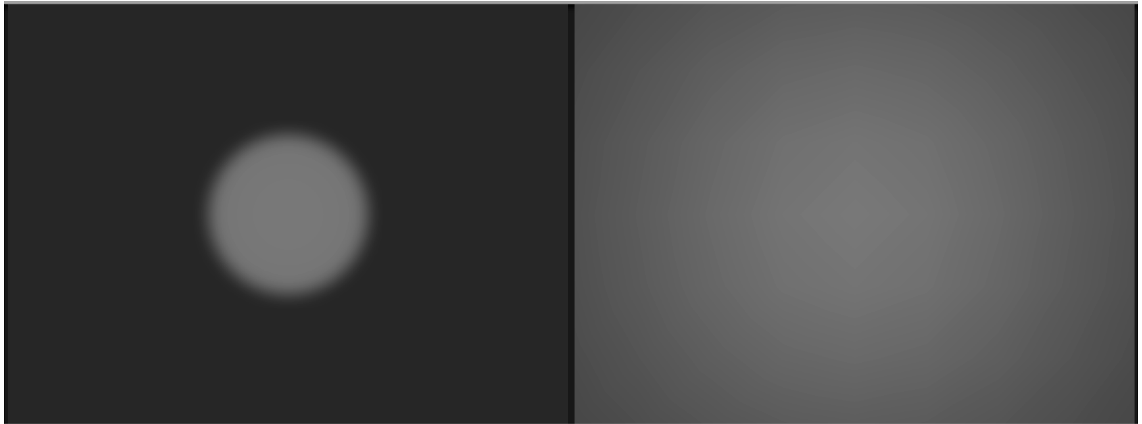
kustannus on ainoastaan valaistujen pikselien määrä.
(Unity Manual 2014d.)

Tässä opinnäytetyössä on käytössä eteenpäin piirretty piirtämispolku, koska jaksotetusti valaistu piirtämispolku kuuluu Unityn Pro-version ominaisuuksiin (Unity Manual 2014d), kun taas verteksivalaistussa piirtämispolussa ei olisi pystytty testaamaan dynaamisen valaistuksen tyyppien vaikutusta pelien suorituskykyyn ollenkaan. Mikäli käytössäsi on jaksotetusti valaistu piirtämispolku, kannattaa pitää mielessä pikselikohtaisen valaistuksen erilaiset kustannukset aina, kun käsitellään pikselikohtaisen valaistuksen optimoimista.

Unityssä on kaksi tapaa tehdä dynaaminen valaistus: pikselikohtainen valaistus ja verteksivalaistus. Jokainen johonkin objektiin vaikuttava pikselikohtaista valaistusta tuottava valonlähde lisää objektin piirtokertojen määrää: jos objektiin vaikuttaa esimerkiksi neljä erillistä valonlähdettä, joissa käytetään pikselikohtaista valaistusta, se joudutaan piirtämään yhteensä viisi kertaa jokaisen ruudunpäivityksen aikana. Tämä luonnollisesti lisää sekä prosessorin että grafiikkaprosessorin kuormitusta, koska objektin piirtämiseksi prosessorin täytyy ensin käsitellä piirtokutsut, minkä jälkeen grafiikkaprosessori käsittelee varsinaisen piirtämisen. Verteksivalaistuksen käyttö sen sijaan ei lisää piirrettävien objektien määrää ollenkaan. (Unity Manual 2014c.) Lisäksi verteksivalaistus on myös laskennallisesti pikselikohtaista valaistusta kevyempi piirtää (Unity Manual 2014e).

Pikselikohtaisen valaistuksen käyttö on kuitenkin joissain tilanteissa perusteltua, koska Point Light -komponentin yksityiskohdat ja Spot Light -komponentin muoto (kuva 6) ovat parempia pikselikohtaisesti piirrettynä. Lisäksi pikselikohtainen valaistus mahdollistaa joitain valaistustehosteita, jotka eivät ole verteksivalaistuksessa mahdollisia. (Unity Manual 2014e.) Pikselikohtaista valaistusta kannattaakin usein käyttää ainoastaan tärkeissä valonlähteissä[3], kun taas vähemmän tärkeissä valonlähteissä verteksivalaistuskin voi olla tarpeeksi näyttävän näköinen (Unity Manual 2014c).

3 Tärkeitä valonlähteitä ovat usein esimerkiksi lähellä olevat valonlähteet tai muita valonlähteitä kirkkaammat valonlähteet (Unity Manual 2014c).

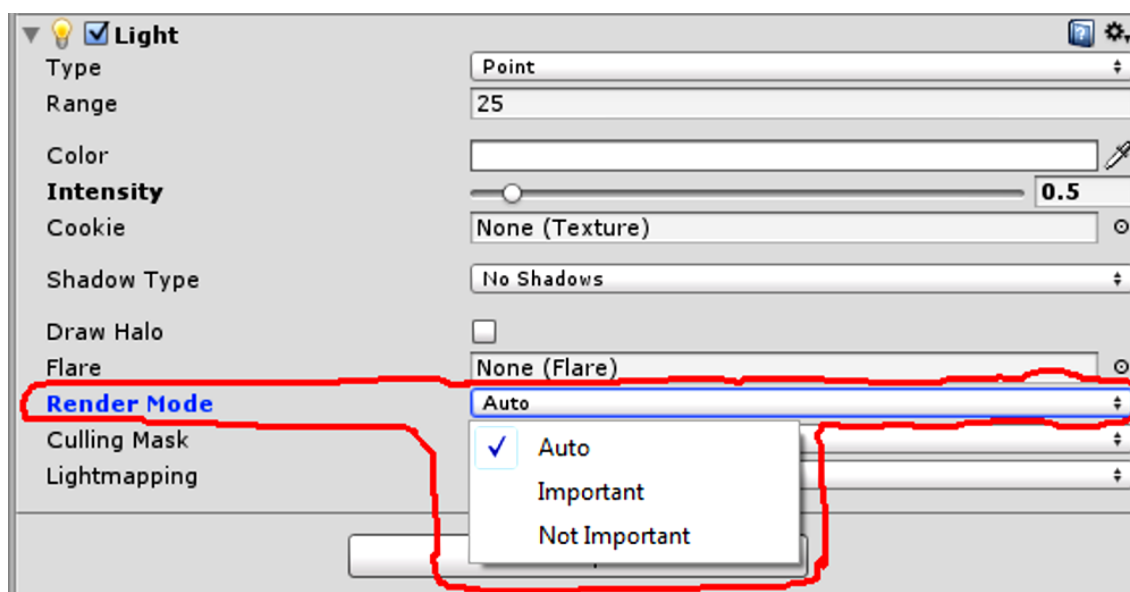


Kuva 6. Kuvakaappauksessa on vasemmalla puolella levy (engl. plane), joka on valaistu Spot Light -komponentilla, jossa käytetään pikselikohtaista valaistusta. Oikealla puolella tämä sama levy on valaistu Spot Light -komponentilla, jossa käytetään verteksivalaistusta.

Dynaamisen valonlähteen tyypin (engl. render mode) voi säätää valo-komponenttien asetuksista (kuva 7):

- Automaattinen (engl. auto) määrittää valonlähteen tyypin ajon aikana riippuen lähellä olevien valojen kirkkaudesta ja laatuasetuksista.
- Tärkeä (engl. important) pakottaa valonlähteen tyypin pikselikohtaiseksi valaistukseksi.
- Ei tärkeä (engl. not important) pakottaa valonlähteen tyypin verteksivalaistukseksi.

(Unity Manual 2014e.)



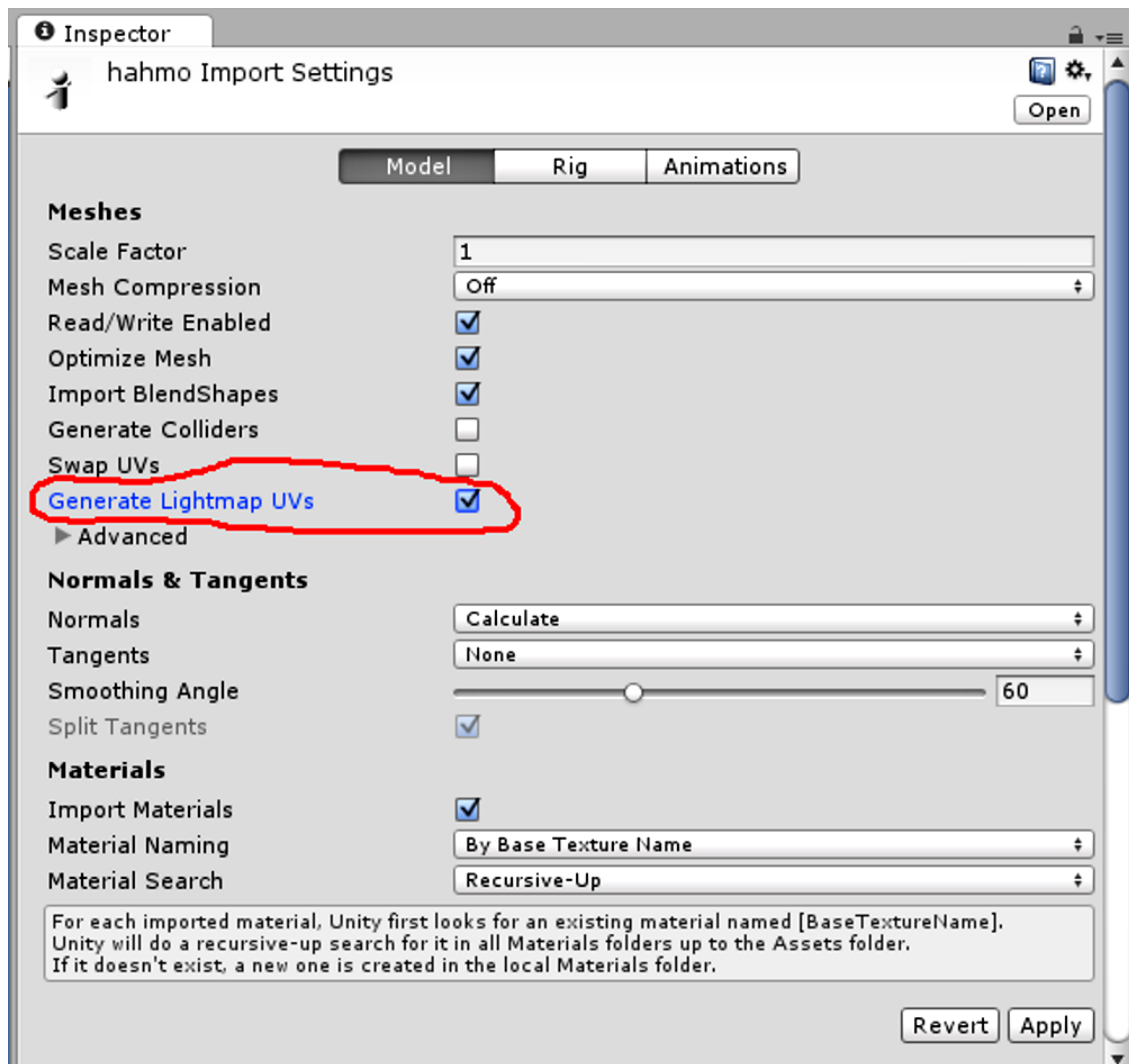
Kuva 7. Kuvakaappaus Point Light -komponentin asetuksista, joissa dynaamisen valonlähteen tyypin vaihtoehdot on korostettu punaisella värillä.

Useat objektit, joita ovat yleensä esimerkiksi seinät, ovat staattisia, eivätkä tällöin tarvitse lainkaan dynaamista valaistusta. Tällöin valaistus on mahdollista leipoa, mikä tosin lisää hieman grafiikkaprosessorin kuormitusta, koska valaistuksen leipominen lisää piirrettävien tekstuurien määrää. (Unity Manual 2014c.)

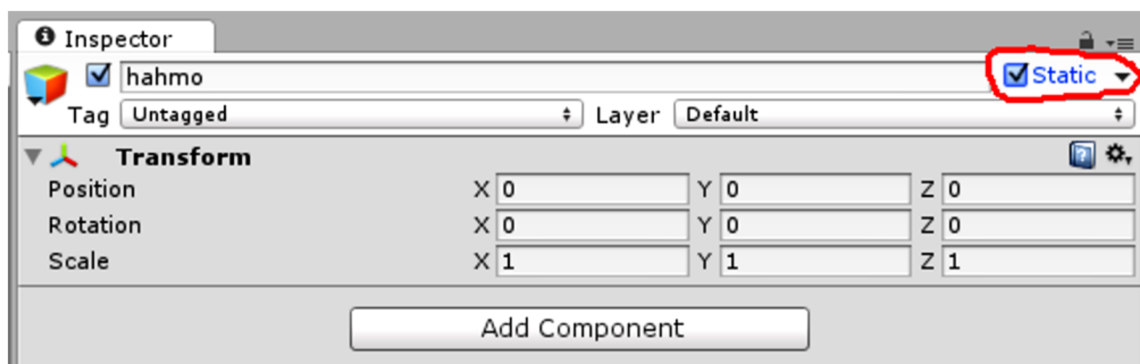
Valaistusta pystyy leipomaan Unityssä tekemällä valokartoituksen, jota varten Unityyn on integroitu valokartoitusliitännäinen. Valokartoituksen tekemistä varten on ensin tehtävä hiukan alkuvalmisteluja:

- Kaikille valokartoitettaville 3D-malleille tulisi olla tehtynä 2D-sovitus, joka on sopiva valokartoitukseen. Niiden tekeminen ei kuitenkaan ole vaikeata, koska Unity osaa generoida ne automaattisesti 3D-mallien tuontiasetuksista (kuva 8).
- Kaikki valokartoitettavat objektit tulisi merkitä staattisiksi, jotta valokartoitusliitännäinen tietäisi, mitkä objektit tulisi valokartoittaa ja mitkä ei (kuva 9).

(Unity Manual 2014f.)



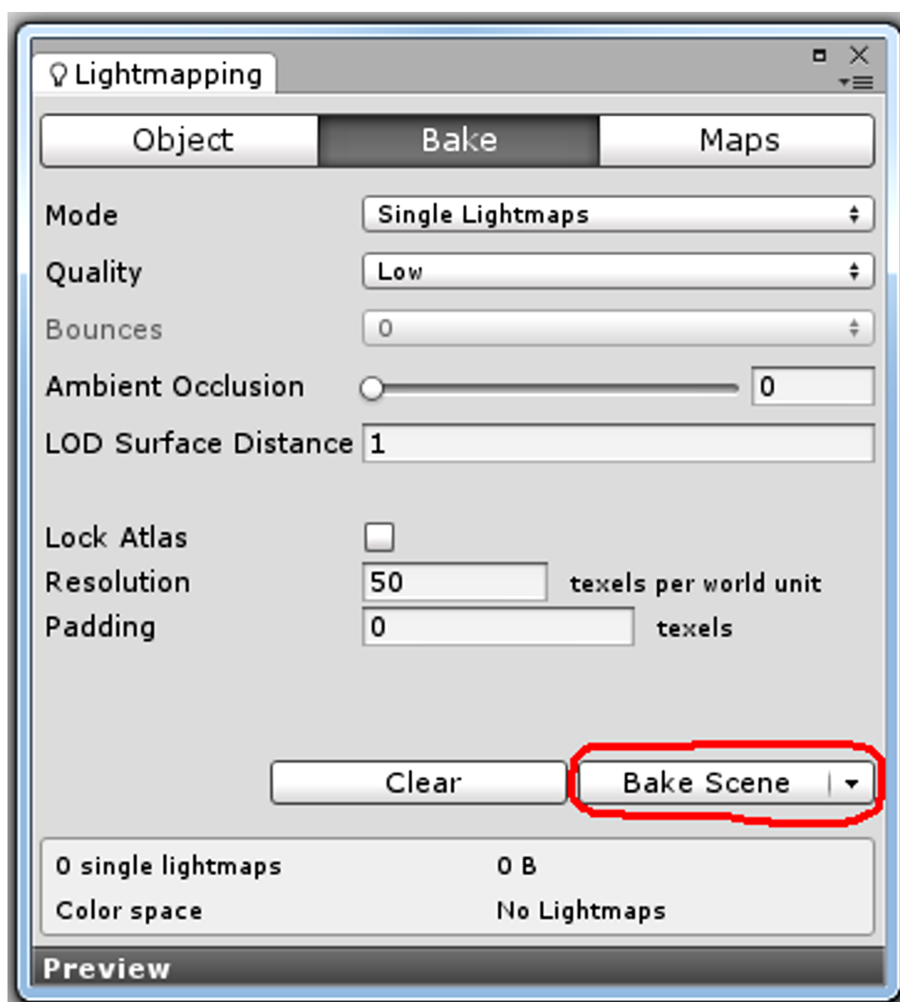
Kuva 8. Kuvakaappaus 3D-mallin tuontiasetuksista, joissa on valittu generoitavaksi 3D-mallista valokartoitukseen sopiva 2D-sovitus. Tämä on korostettu punaisella värillä.



Kuva 9. Kuvakaappauksessa merkitään 3D-malli staattiseksi valokartoitusta varten, mikä on korostettu punaisella värillä.

Tämän jälkeen valokartoitusliitännäisestä (kuva 10) voi vielä halutessaan säätää erilaisia asetuksia, joista grafiikoiden suorituskyvyn kannalta merkityksellisiä

ovat valokartoituksen tyyppi (engl. mode) ja resoluutio, koska ne vaikuttavat siihen, kuinka monta valokartta-tekstuuria generoidaan. (Unity Manual 2014g.)



Kuva 10. Kuvakaappaus valokartoitusliitännäisestä, jossa on korostettu punaisella värillä Bake Scene -painike. Sitä painamalla leipominen käynnistyy mikäli alkuvalmistelut on tehty oikein.

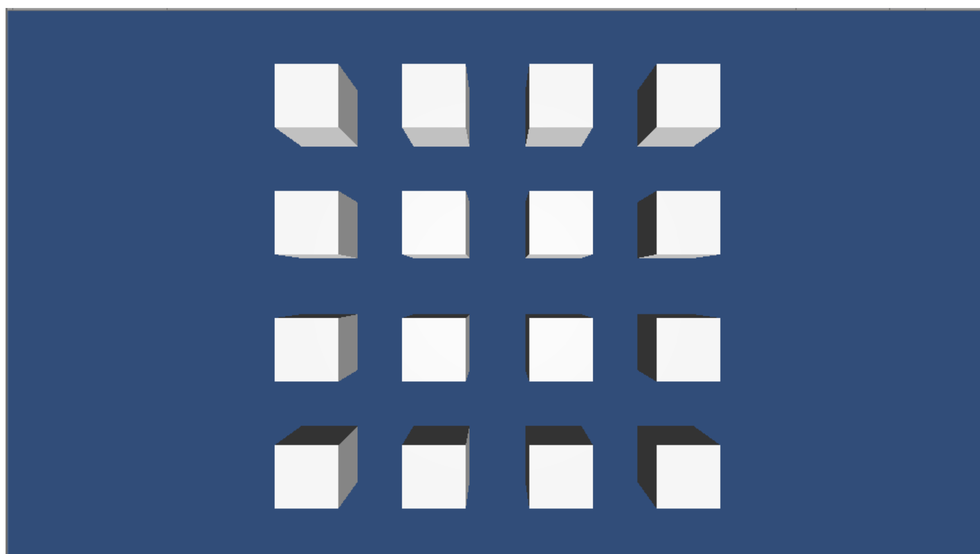
Valokartoituksen tyyppejä on kolme erilaista: yksinkertainen valokartoitus (engl. single lightmaps), kaksinkertainen valokartoitus (engl. dual lightmaps) ja kohdistettu valokartoitus (engl. directional lightmaps). Tässä opinnäytetyössä käytetään yksinkertaista valokartoitusta, koska kaksinkertainen valokartoitus tarvitsisi toimiakseen jaksotetusti valaistun piirtämispolun tai vaihtoehtoisesti sitä varten pitäisi ohjelmoida omat varjostimensa, mikä olisi ollut liian työlästä tehdä tämän opinnäytetyön puitteissa. Kohdistettu valokartoitus sen sijaan on täysin Unityn Pro-version ominaisuus. (Unity Manual 2014g.)

Keskeisin ero yksinkertaisen valokartoituksen ja kahden muun valokartoituksen tyypin välillä on se, että yksinkertaisessa valokartoituksessa luodaan ainoastaan yhdet valokartat, joita käytetään kameran kaikille etäisyyksille, kun taas muissa tyypeissä luodaan erikseen valokartat kameran lähi- ja kaukoetäisyyksille. Yksinkertaiselle valokartoitukselle luotavat valokartat ovat tismalleen samat valokartat, jotka muille valokartoituksen tyypeille luotaisiin kameran kaukoetäisyyksille. (Unity Manual 2014g.)

4.1.2 Kameran asettelu

Kamera osaa jättää piirtämättä ne objektit, jotka jäävät kameran näkemän alueen ulkopuolelle. Tämä toiminnallisuus perustuu kameran OnBecameVisible -ja OnBecameInvisible-funktioihin, joita kamera osaa kutsua automaattisesti tarpeen tullen ilman, että niille tarvitsee itse tehdä mitään. (Unity Scripting API 2014b; 2014c.)

Tätä toiminnallisuutta voi käyttää hyödyksi optimoinnissa: esimerkiksi kameran tuominen lähemmäksi pelialuetta voi rajata useita objekteja kameran näkemän alueen ulkopuolelle. Koska objekteja piirretään tällöin vähemmän, siitä on hyötyä prosessorin lisäksi grafiikkaprosessorille, aivan kuten pikselikohtaisen valaistuksen vähentämisenkin kanssa. Kameran asettelua on havainnollistettu kuvilla 11 ja 12, joista molemmat ovat kuvakaappauksia tasosta, jossa on 16 kuutiota, mutta kuvassa 11 kamera on kaukana ja kuvassa 12 lähellä.



Kuva 11. Kuvakaappauksessa kamera on kaukana, minkä vuoksi kaikki 16 kuutiota piirretään.

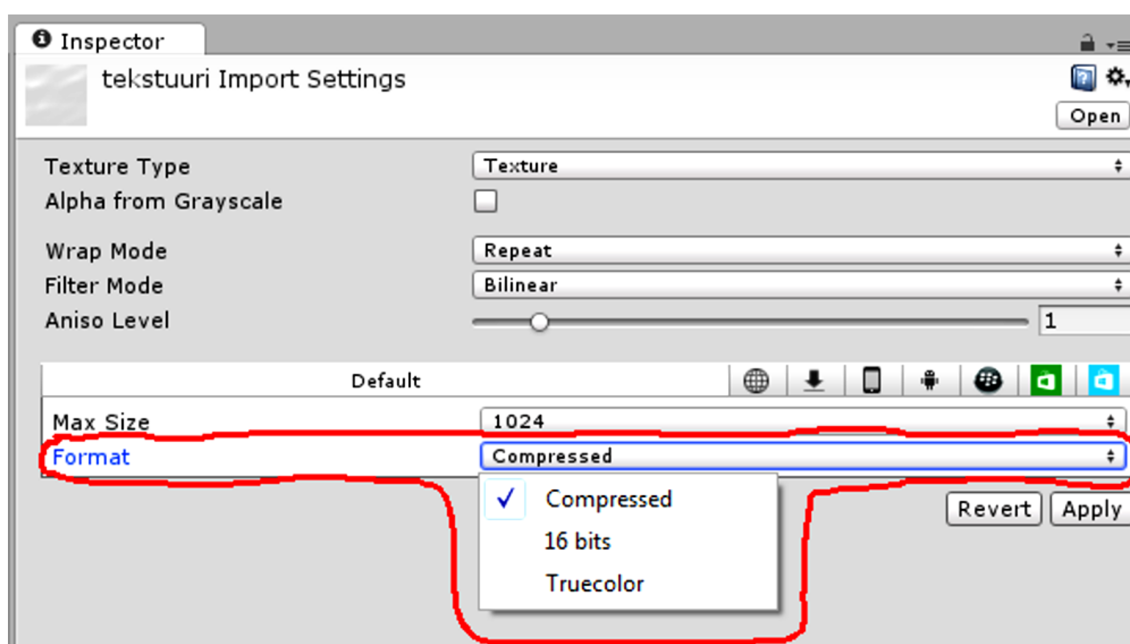
Kuva 12. Kuvakaappauksessa kamera on lähellä, minkä vuoksi ainoastaan neljä kuutiota piirretään.

4.2 Grafiikkaprosessorin suorituskyvyn parantaminen

Siinä missä prosessorin keskeisin tehtävä on käsitellä piirtokutsut, grafiikkaprosessori käsittelee varsinaisen piirtämisen. Tämän vuoksi tekstuurien tarkkuus ja kerroksellisuus vaikuttavat suoraan grafiikkaprosessorin suorituskykyyn. (Unity Manual 2014c.)

4.2.1 Tekstuurien optimointi

Tekstuureita voi optimoida pakkaamalla niitä, jolloin tekstuurien koko pienenee ja grafiikkaprosessorin työmäärä vähenee (Unity Manual 2014c). Tekstuurien pakkaaminen on hyödyllistä myös, mikäli pelin lopullisesta tiedostokoosta halutaan mahdollisimman pieni, sillä tekstuurit ovat yleisin pullonkaula pelien tiedostojen koossa (Unity Manual 2014b). Tekstuurien pakkaaminen tehdään tekstuurien tuontiasetuksista (kuva 13), joista valittavina on erilaiset pakkausvaihtoehdot riippuen tekstuurin tyypistä (Unity Manual 2014a).



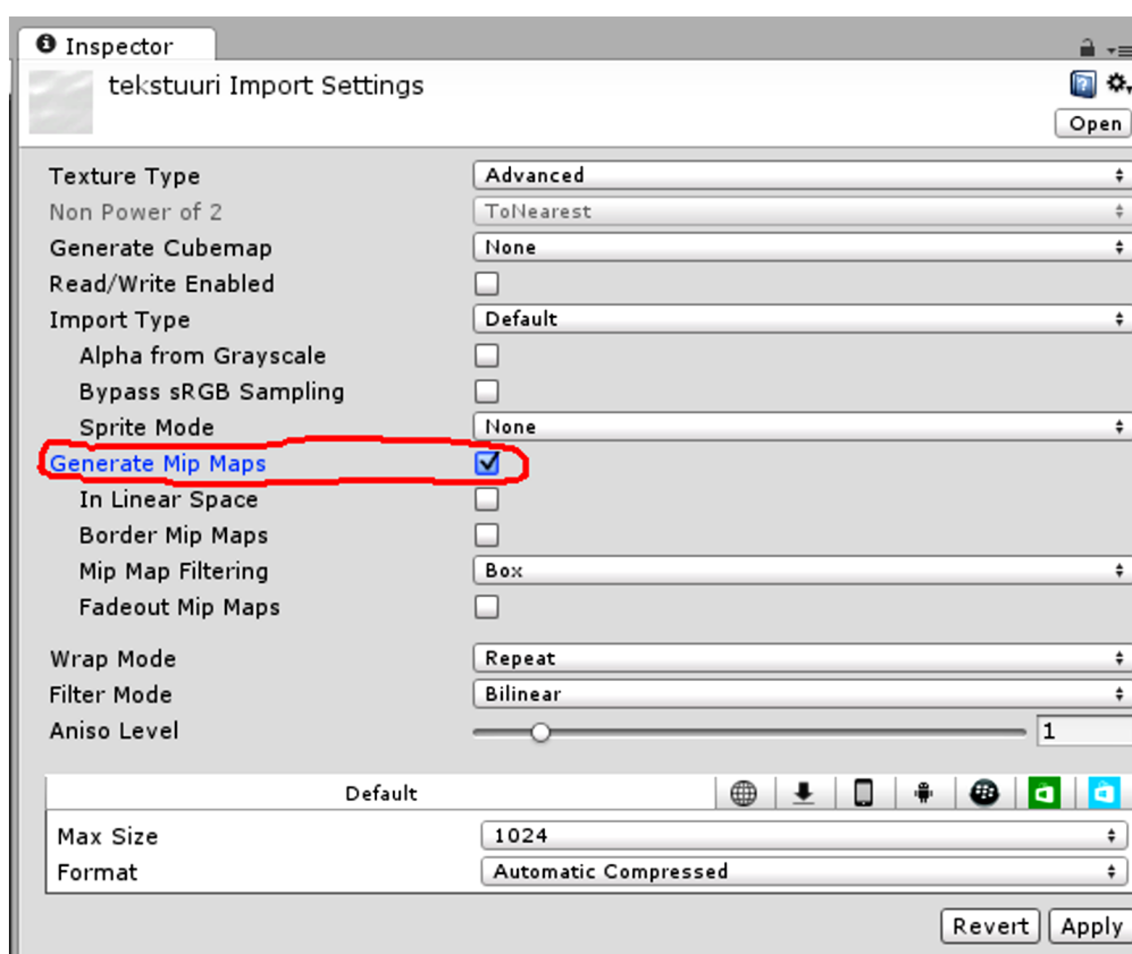
Kuva 13. Kuvakaappaus tavanomaisimman tyyppisen tekstuurin tuontiasetuksista, joista nähdään punaisella korostettuna, että tekstuurin tyypiksi (engl. format) voi valita pakatun (engl. compressed) tai pakkaamattoman (kaksi vaihtoehtoa: 16 bits ja truecolor).

3D-mallien tekstuureita[4] voi lisäksi optimoida tekemällä niille Mip-kartoituksen (Unity Manual 2014c). Unityn manuaalissa (2014a) kerrotaan, että Mip-kartoituksen haittapuolena tekstuuritiedoston koko kasvaa 33 %, mutta Mip-kartoituksesta on manuaalin mukaan silti ehdottomasti enemmän hyötyä kuin haittaa.

4 3D-malleille tarkoitettuja tekstuureita (jotka ovat siis todellisuudessa 2D-tekstuureita) ei tule sekoittaa varsinaisiin 3D-tekstuureihin, joita voi luoda Unityssä toistaiseksi ainoastaan skriptaamalla (Unity Manual 2014h).

Mobiilipeleihin erikoistunut pelinkehittäjä Mel Georgiou on tästä asiasta kuitenkin toista mieltä: hän kertoo blogissaan (2012), että joissain tilanteissa Mip-kartoituksesta voi olla enemmän haittaa kuin hyötyä, mitä hän havainnollistaa Flight Unlimited Las Vegas (Flight Systems LLC 2012) -mobiilipelistä otetulla kaupungilla. Sen tekstuurien optimoinnissa parhaaseen lopputulokseen päästiin, kun Mip-kartoitusta oli käytetty vain osissa 3D-mallien tekstuureista.

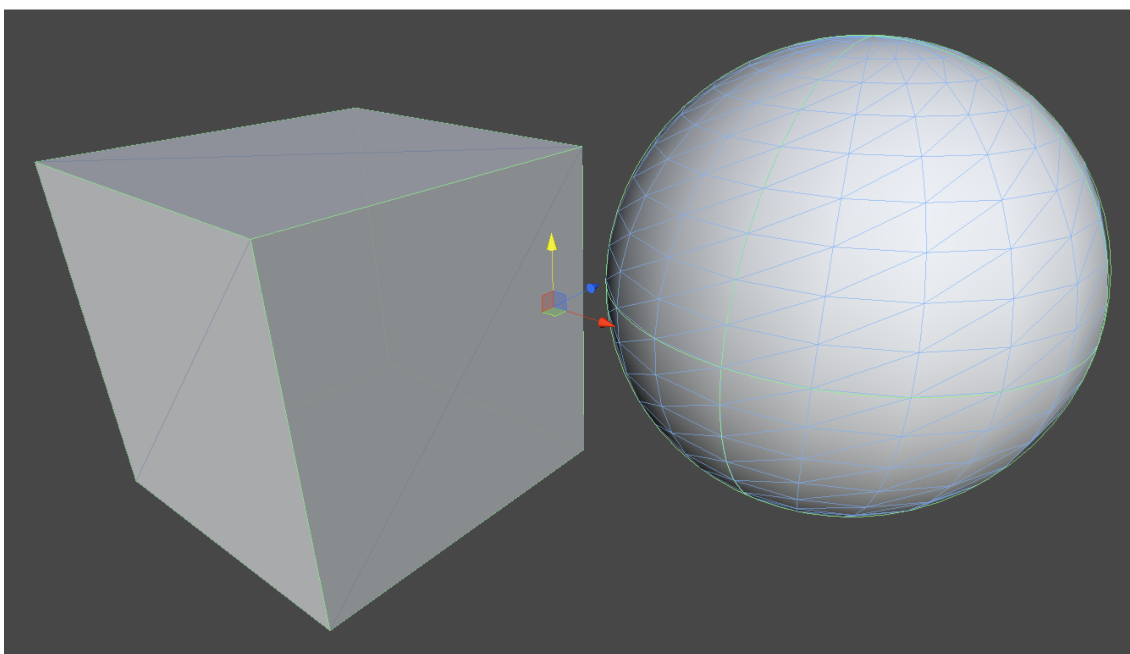
Unity osaa generoida Mip-kartoituksen automaattisesti: pelin kehittäjän tarvitsee ainoastaan sallia sen käyttö tekstuurien tuontiasetuksista (kuva 14).



Kuva 14. Kuvakaappauksessa 3D-mallille tarkoitetun kehittyneemmän tekstuurin tuontiasetuksista on valittu generoitavaksi Mip-kartoitus, mikä on korostettu punaisella värillä.

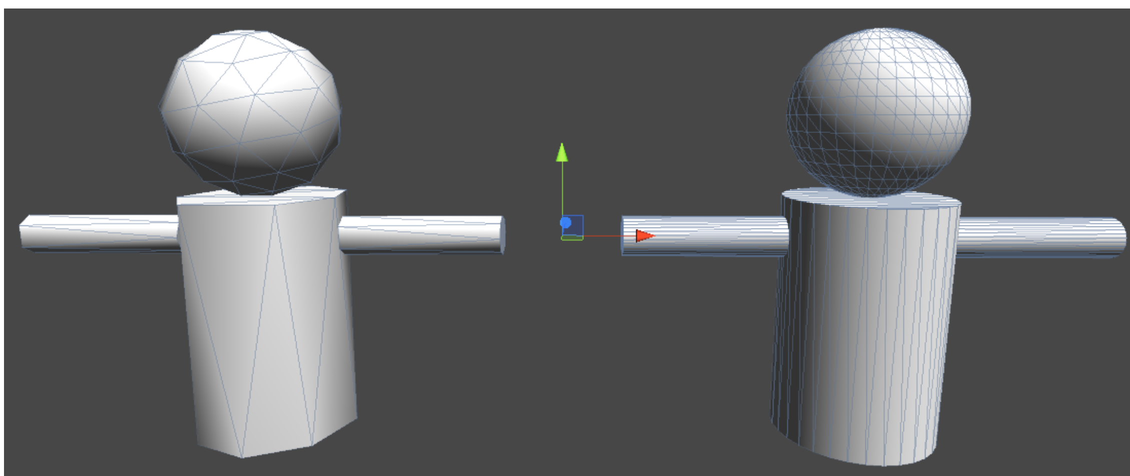
4.2.2 Kolmioiden määrän optimointi

Tekstuurien tarkkuuden ja kerroksellisuuden lisäksi myös kolmioiden määrä vaikuttaa grafiikkaprosessorin suorituskykyyn, minkä vuoksi 3D-mallien muoto kannattaa pitää mahdollisimman yksinkertaisena: esimerkiksi kuution piirtäminen on kevyempi työ kuin pallon piirtäminen, koska kuutiossa piirrettäviä kolmioita on huomattavasti palloa vähemmän (kuva 15). 3D-mallien muodon yksinkertaistaminen lisää grafiikkaprosessorin lisäksi myös hieman prosessorin suorituskykyä. (Unity Manual 2014c.)

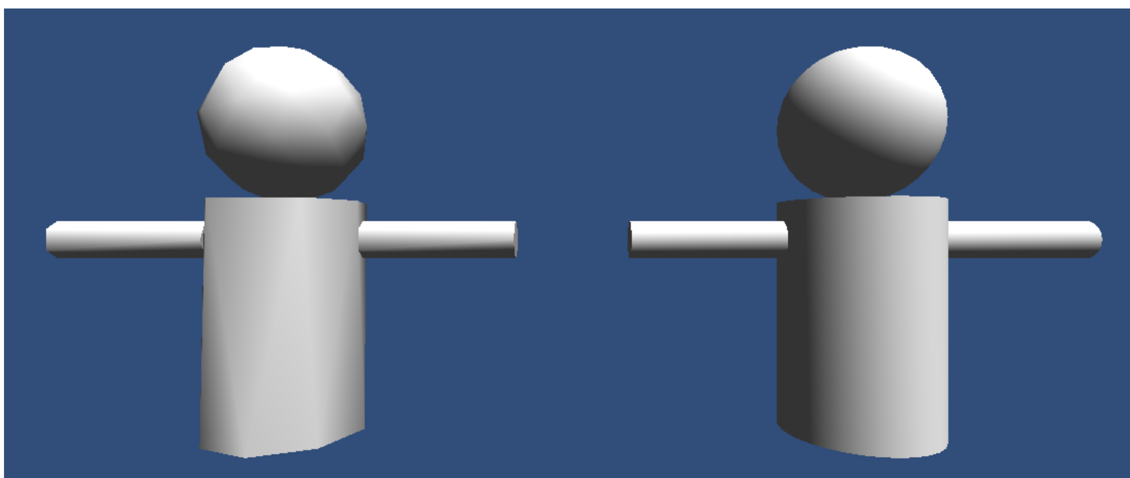


Kuva 15. Kuvakaappaus kuutiosta ja pallosta Unityn editointi-tilassa, josta näkee selkeästi kuinka pallossa on moninkertainen määrä kolmioita verrattuna kuutioon.

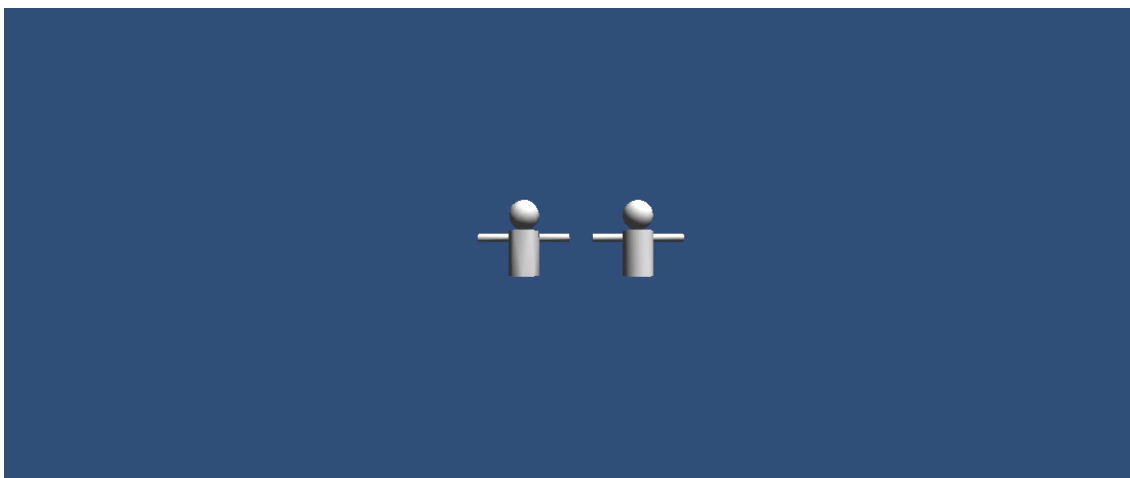
Se, kuinka paljon kolmioita kussakin 3D-mallissa on järkevää käyttää, riippuu ensinnäkin siitä mitä haetaan: joissain peleissä 3D-mallien kulmikkuus ei haittaa (kulmikkaissa 3D-malleissa on luonnollisesti vähemmän kolmioita kuin pyöreissä) tai se voi olla jopa tyylikeino. Toiseksi, myös kameran etäisyydellä kuvatuista 3D-malleista on merkitystä: mikäli kamera on hyvin kaukana, kolmioita tarvitaan usein vähemmän, kun taas läheltä kuvattuihin 3D-malleihin niitä tarvitaan usein enemmän. Kameran etäisyyden vaikutusta kolmioiden määriin on havainnollistettu kuvilla 16, 17 ja 18.



Kuva 16. Kuvakaappauksessa nähdään kaksi hahmoa, joista vasemmanpuoleinen koostuu 180:stä ja oikeanpuoleinen 1332:sta kolmiosta. Hahmot nähdään Unityn editointi-tilassa, jotta kolmioiden määrän eroavaisuudet näkee parhaiten.



Kuva 17. Kuvakaappauksessa nähdään samat hahmot kuin kuvassa 16 ajon aikana kameran ollessa lähellä hahmoja, jolloin vasemmanpuoleinen hahmo näyttää selkeästi oikeanpuolimmaista kulmikkaammalta.



Kuva 18. Kuvakaappauksessa nähdään samat hahmot kuin kuvissa 16 ja 17 ajon aikana kameran ollessa kaukana hahmoista, jolloin hahmot näyttävät kutakuinkin identtisiltä.

5 Muut optimointikeinot

Tässä luvussa esitellään sekalaisia Unityn optimointikeinoja, jotka eivät liity pelien grafiikoihin.

5.1 Tarpeettomien päivitysten välttäminen

Jokaisella ruudunpäivityksellä tehtäviä päivityksiä tulisi välttää, mikäli vähempikin riittää (Unity Scripting Reference). Tämän vuoksi on syytä tutustua siihen, miten Unityn sisäänrakennetut funktiot toimivat:

- Awake- ja Start-funktiot kutsutaan ainoastaan yhden kerran tason alussa. Ainut ero näiden funktioiden välillä on se, että Awake-funktiot kutsutaan aina ennen Start-funktioita. Awake- ja Start funktioita voi käyttää esimerkiksi muuttujien määrittelyyn. (Unity Tutorials 2014.)
- Update-funktio kutsutaan kerran jokaisen ruudunpäivityksen aikana, ja sitä voi käyttää minkä tahansa toiminnallisuuden toteuttamiseen. (Unity Scripting API 2014d.)
- FixedUpdate-funktio kutsutaan kerran jokaisen muunnetun ruudunpäivi-

tyksen (engl. fixed framerate frame) aikana, ja sitä suositellaan käytettäväksi erityisesti jäykkien kappaleiden kanssa (Unity Scripting API 2014e).

- OnGUI-funktio voi tulla kutsutuksi useita kertoja yhden ruudunpäivityksen aikana, ja sitä käytetään esimerkiksi erilaisten valikoiden tekemiseen (Unity Scripting API 2014f).
- Coroutine-funktio kutsutaan tietyn intervallin välein, jonka ohjelmoija voi itse päättää. Coroutinea kannattaa käyttää aina, kun se on mahdollista. (Unity Scripting Reference.)
- Laukaisin (engl. trigger) kutsutaan jonkin tietyn tapahtuman seurauksena (esimerkiksi 2 objektia osuu toisiinsa), ja niitä kannattaa käyttää aina, kun se on mahdollista (Unity Scripting Reference).

Unityn funktioiden optimaalista käyttöä on havainnollistettu esimerkkikuvin: kuvissa 19 ja 20 määritellään ensin kohde-nimisen 3D-vektorin sijainti[5], minkä jälkeen objektia siirretään kohti kohdetta. Molemmissa kuvissa toiminnallisuus on täysin sama, mutta kuvassa 19 kohde määritellään Update-funktion sisällä, kun taas kuvassa 20 kohde määritellään Start-funktion sisällä. Objektin liikuttaminen tehdään molemmissa kuvissa Update-funktion sisällä.

Tämä tarkoittaa sitä, että kuvassa 19 kohde määritellään uudestaan jokaisella ruudunpäivityksellä, kun taas kuvassa 20 se tehdään ainoastaan kerran tason alussa. Koska kohde tarvitsee määritellä ainoastaan kerran, sen määrittely on järkevämpi suorittaa Start-funktion sisällä, kuten kuvassa 20. Kuvissa 19 ja 20 ohjelmointikielenä on C#, mutta sama periaate myös muihin Unityn tukemiin ohjelmointikieliin[6].

5 Matematiikassa vektoreilla on ainoastaan suuruus ja suunta, mutta Unityssä vektoreilla voi myös ilmaista jonkin pisteen sijainnin (Unity Scripting API 2014g; 2014h).

6 Unity tukemat ohjelmointikielet ovat C#, UnityScript ja Boo, joista C# on tällä hetkellä ylivoimaisesti suosituin ohjelmointikieli (Aleksandr 2014).

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LiikutaObjektia : MonoBehaviour {
5
6      Vector3 kohde;
7
8      void Start () {
9
10     }
11
12     void Update () {
13         // Määritellään kohde-nimisen 3D-vektorin sijainti:
14         kohde = new Vector3(5, 0, 0);
15
16         // Liikutetaan objektia kohteen sijaintia kohti:
17         transform.position = Vector3.MoveTowards(transform.position, kohde, 10 * Time.deltaTime);
18     }
19 }
```

Kuva 19. Kuvakaappauksessa sekä kohde-nimisen 3D-vektorin sijainnin määrittely että objektin liikuttaminen kohti kohdetta suoritetaan Update-funktion sisällä.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LiikutaObjektia : MonoBehaviour {
5
6      Vector3 kohde;
7
8      void Start () {
9          // Määritellään kohde-nimisen 3D-vektorin sijainti:
10         kohde = new Vector3(5, 0, 0);
11     }
12
13     void Update () {
14         // Liikutetaan objektia kohteen sijaintia kohti:
15         transform.position = Vector3.MoveTowards(transform.position, kohde, 10 * Time.deltaTime);
16     }
17 }
18
```

Kuva 20. Kuvakaappauksessa kohde-nimisen 3D-vektorin sijainnin määrittely suoritetaan Start-funktion sisällä ja ainoastaan objektin liikuttaminen kohti kohdetta suoritetaan Update-funktion sisällä.

5.2 Objektien altaus

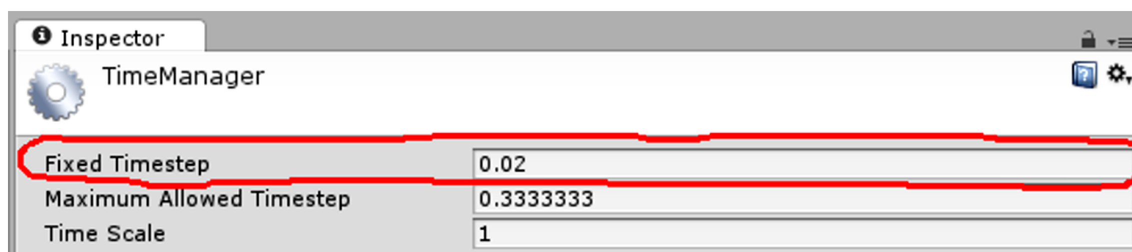
Jokainen kerta, kun objekti luodaan, sille varataan muistia, ja kun se tuhotaan, muistia vapautetaan. Jatkuva muistin varaaminen ja vapauttaminen on yleensä vähemmän optimaalista, kuin että muistia olisi kokoajan käytössä jokin vakiomäärä. Siihen on kaksi syytä: ensinnäkin, jokainen kerta, kun objekti luodaan tai tuhotaan, se rasittaa roskien kerääjää, ja toiseksi objekteja luodessa joudutaan aina lataamaan objektiin liittyvät komponentit, kuten esimerkiksi tekstuurit. (Unity Manual 2012c.)

Objektien jatkuvaa muistin varaamista ja vapauttamista voidaan vähentää käyttämällä objektien altausta. Objektien altaus on erityisen optimaalinen ratkaisu objekteille, joiden elinikä on lyhyt ja joita tarvitaan usein (esimerkiksi aseen ammukset). (Unity Manual 2012c.)

Objektien altauksella voi olla myös kielteisiä vaikutuksia pelien suorituskykyyn, mikäli objektien altaus on käytössä ja sen lisäksi luodaan runsaasti uusia objekteja. Tällöin roskien kerääjää joudutaan kutsumaan normaalia enemmän, ja lisäksi se toimii hitaammin, koska aktiivisena on yhtäaikaisesti paljon objekteja. Ylipäänsä objektien pitämistä aktiivisena kannattaa välttää, mikäli objekteja ei enää tarvita. (Unity Manual 2012c.)

5.3 Fysiikoiden optimointi

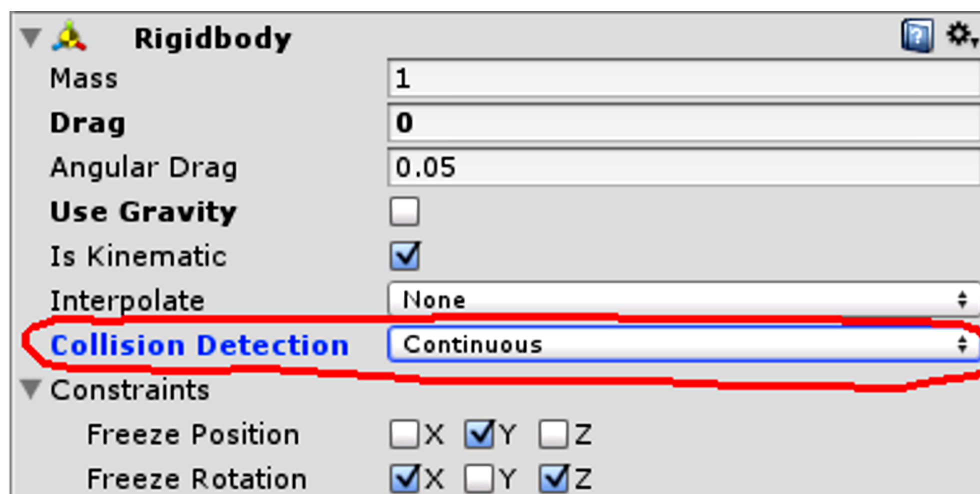
Fysiikoita voi optimoida nostamalla muunnettun aika-askeleen arvoa: fysiikoita ei välttämättä tarvitse päivittää läheskään jokaisella ruudunpäivityksellä. Se, miten muunnetun aika-askeleen arvo vaikuttaa fysiikoiden päivitysten määrään, lasketaan jakamalla luku 1 muunnetun aika-askeleen arvolla. (Unity Manual 2014i.) Eli mikäli muunnetun aika-askeleen arvoksi asetetaan esimerkiksi 0.1, fysiikoita päivitetään 10 kertaa sekunnissa (1/0.1=10). Muunnettuja aika-askeleita voi säätää Unityn ajanhallinta-asetuksista (engl. time manager, kuva 21).



Kuva 21. Kuvakaappaus Unityn ajanhallinta-asetuksista, jossa on korostettu punaisella värillä muunnetun aika-askeleen arvo, joksi on asetettu 0.02. Fysiikoita päivitetään tällöin 50 kertaa sekunnissa.

Mikäli muunnetun aika-askeleen arvoa nostaa liikaa, siinä on riskinä, että objektit voivat mennä toistensa lävitse, vaikka objekteilla olisikin asianmukaiset tör-

mäyttimet. Tämä johtuu siitä, että edellinen fysiikanpäivitys voidaan suorittaa, kun objekti on törmäyttimen toisella puolella ja seuraava vasta, kun objekti on jo päässyt törmäyttimen vastakkaiselle puolelle. Tätä varten jäykkien kappaleiden asetuksissa on mahdollista asettaa törmäyksien tunnistaminen jatkuvaksi (kuva 22). (Unity Manual 2014j.)



Kuva 22. Kuvakaappaus jäykän kappaleen asetuksista, jossa törmäyksien tunnistus on asetettu jatkuvaksi.

Fysiikoita voi optimoida myös yksinkertaistamalla törmäyttimiä, joten esimerkiksi 3D-mallien verkon myötäisiä törmäyttimiä (engl. mesh collider) kannattaa välttää, kun taas kaikkein yksinkertaisimpia törmäyttimiä, laatikkotörmäyttimiä (engl. box collider), kannattaa suosia. Tämän lisäksi staattisten objektien törmäyttimiin voidaan käyttää staattisia törmäyttimiä, mikä tarkoittaa sitä, että objekteihin liitetään törmäytin ilman jäykkää kappaletta. Tällöin fysiikkamoottori pystyy optimoimaan hieman törmäyttimien toimintaa, mikä perustuu siihen oletukseen, että törmäytin ei liiku. Mikäli staattisen törmäyttimen laittaa dynaamiseen objektiin, siitä päinvastaisesti aiheutuu haittaa fysiikkamoottorin suorituskyvylle, minkä lisäksi siitä voi aiheutua bugeja peliin. (Unity Manual 2014k.)

Lisäksi fysiikoita optimoidessa tulee huomioida se, että onko kyseessä kaksi- vai kolmiulotteinen peli: Unityssä on erikseen 2D- ja 3D-peleille optimoidut jäykät kappaleet ja törmäyttimet (Unity Manual 2014j; 2014k; 2014l).

6 Optimoitavat Unity-pelit

Aivan opinnäytetyöni tekemisen alussa kyselin sähköpostitse Karelia-ammattikorkeakoulun opiskelijoilta, josko saisin joitain Unity-pelejä optimoitavaksi. Sain yhden myöntävän vastauksen projektitiimiltä, joka oli ollut tekemässä prototyyppiä Icemare-tasohyppelypeliin. Olin ollut myös itse tekemässä kahta Unity-pelin prototyyppiä: Puck Buddies -jääkiekkopeliä ja Sneak Challenge -hiiviskelypeliä. Lisäksi olin itsekseni kehittänyt Status: Insane -projektinimellä omaa hiiviskelypeliäni.

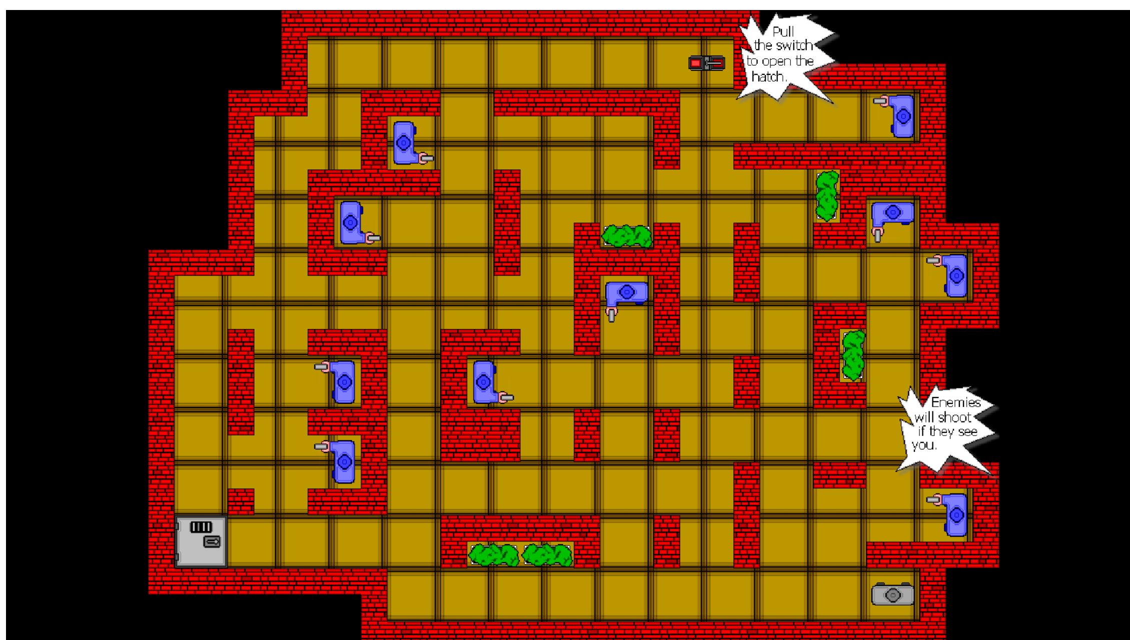
Koska hiiviskelypelejä olisi ollut optimoitavaksi jopa kaksi kappaletta, päätettiin Sneak Challenge jättää pois, koska Status: Insane oli sitä hiukan laajempi peli. Tämän lisäksi valintaan vaikutti se, että Status: Insane oli saatavissa olleista peleistä ainut, joka oli käännetty PC:n lisäksi myös Androidille, ja uskoisin, että mobiilipelin saaminen mukaan toi lisäarvoa tämän opinnäytetyön käytännön osioon mobiililaitteiden resurssien niukkuuden vuoksi. Lisäksi testaaminen useammalla alustalla mahdollisti sen, että pystyttiin tekemään myös alustakohtaisia vertailuja sen sijaan, että olisi tehty pelkästään pelikohtaisia vertailuja.

Täten optimoitavaksi olisi ollut kolme hyvin erilaista Unity-peliä, joista jokainen olisi edustanut täysin eri genreä. Aika ei kuitenkaan riittänyt kuin kahden pelin testaamiseen, minkä vuoksi Icemare jätettiin pois testattavista peleistä, mutta jäljelle jääneistä kahdesta pelistäkin saatiin kerättyä varsin paljon tietoa. Syy, miksi Icemare jätettiin pois Puck Buddiesin sijaan, oli se, että Puck Buddies vaikutti olevan Icemarea hieman valmiimpi peli, sillä Icemaresta näytti puuttuvan useita tärkeitä ominaisuuksia, eikä siinä ollut esimerkiksi yhtään valmista tasoa. Molemmat testattavaksi päätyneet pelit oli toteutettu Unityn ilmaisella versiolla, joten mitään yhteensopivuusongelmia ei sen vuoksi ollut.

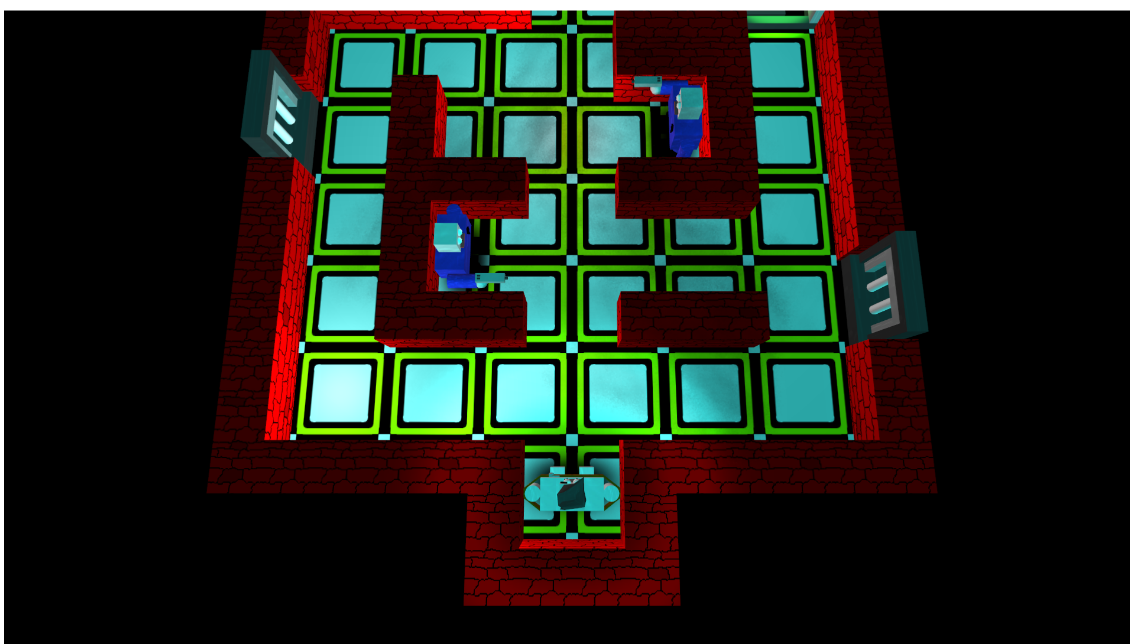
6.1 Status: Insane

Status: Insane (työnimi) on kehittämäni hiiviskelypeli, joka eroaa useimmista muista hiiviskelypeleistä, kuten esimerkiksi Tom Clancy's Splinter Cell -pelisarjan (Ubisoft 2002–2013) peleistä siten, että vartijoiden tekoäly on tässä pelissä todella yksinkertainen: vartijat osaavat ainoastaan joko seistä paikoillaan tai kääntyä tietyn intervallin välein, minkä lisäksi ne osaavat ampua pelaajan ohjaaman pelihahmon nukutusnuolella tämän nähdessään. Vartijoilla on myös todella rajoittunut näkökyky: vartijoiden näkökenttä on suoran putken muotoinen. Vartijoiden yksinkertaista tekoälyä ja rajoittunutta näkökykyä on kompensoitu lisäämällä vartijoiden määrää: yhdessä ahtaassa tasossa voi olla jopa yli kymmenen vartijaa, eikä heistä yksikään saa nähdä pelaajaa tai muuten tason joutuu aloittamaan alusta. Tämän vuoksi pelissä onkin kaikesta huolimatta jonkin verran haastetta.

Pelin prototyyppi tehtiin Game Makerilla (YoYo Games 2011) keväällä 2012, jolloin peli oli vielä 2D-peli (kuva 23). Pelin Unity-version (kuva 24) kehitys aloitettiin tammikuussa 2013, jolloin peliin päätettiin samalla lisätä kolmas ulottuvuus. Pelin tämän hetkinen versio saatiin valmiiksi elokuussa 2013, ja sen alustoina ovat PC sekä Android. Pelin kehitystä on tarkoitus jatkaa mahdollisesti joskus myöhemmin.



Kuva 23. Kuvakaappaus Status: Insanen prototyypistä, joka tehtiin Game Maker -pelimoottorilla.



Kuva 24. Kuvakaappaus Status: Insanen Unity-version ensimmäisestä tasosta. Tässä kuvakaappauksessa sen alustana on PC.

6.2 Puck Buddies

Puck Buddies (kuva 25) on Karelia-ammattikorkeakoulun ja Outokummun peli-opetuksen opiskelijoiden yhteistyönä syntynyt prototyyppi sovelletusta 3D-jää-

kiekkopelistä, jota myös minä olin tekemässä. Vastuualueisiini kuului mm. pelin ääniefektit, kaikki ääniin liittyvät ohjelmoinnit ja aloitusten voittajan ratkaisevan reaktiotesti-minipelin ohjelmointi (kuva 26).

Kuva 25. Kuvakaappaus Puck Buddiesista.

Kuva 26. Kuvakaappaus Puck Buddiesista aloitustilanteessa, jossa aloituksen voittaja ratkaistaan reaktiotesti-minipelillä, jossa täytyy painaa oikeita peliohjaimen painikkeita tarpeeksi nopeasti.

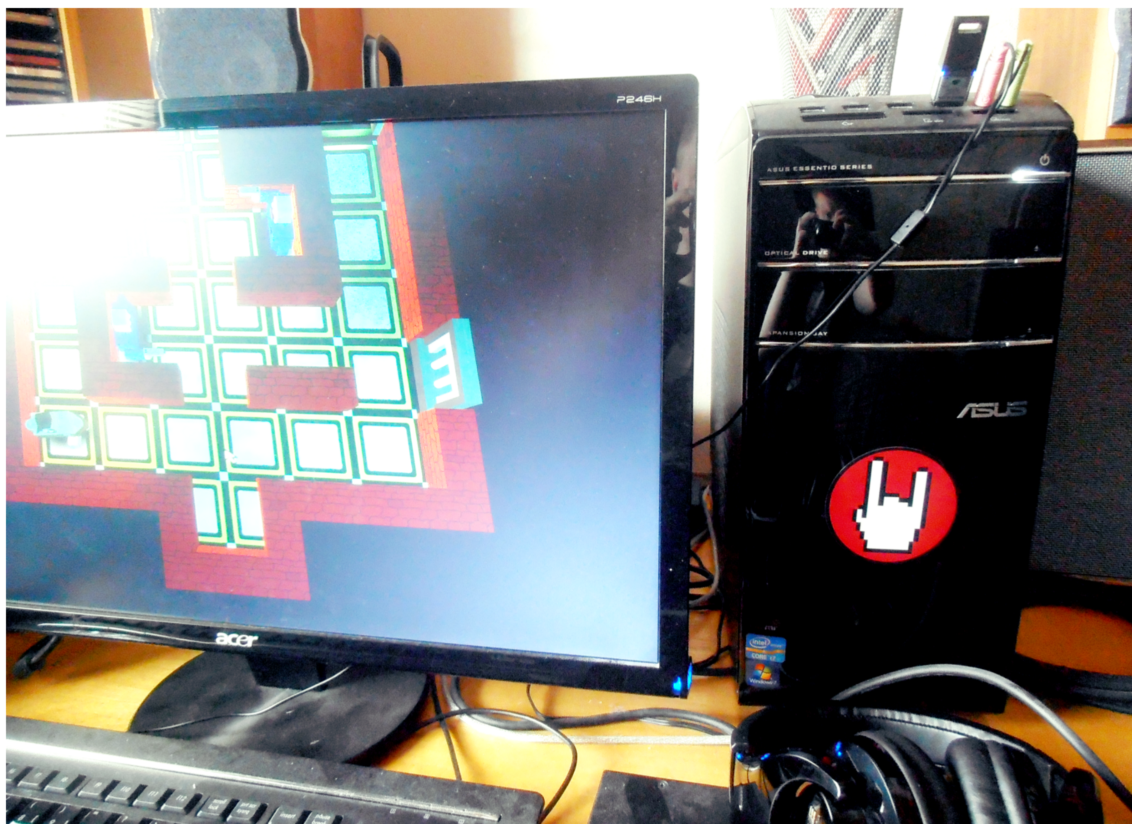
Peli eroaa esimerkiksi NHL-pelisarjan (EA Sports 1991–2014) peleistä siten, että tässä pelissä on vähemmän kenttäpelaajia, eikä peli pyri olemaan kovinkaan realistinen. Prototyyppi tehtiin vuoden 2013 keväänä ja sen alustana on ainoastaan PC.

7 Testilaitteet

Tässä opinnäytetyössä testattavien pelien alustoina ovat PC ja Android, jotka edustavat videopelaamisen vastakkaisia ääripäitä: markkinoiden tehokkaimmat laitteet ovat yleensä PC-tietokoneita, kun taas mobiililaitteet, joihin myös testilaitteena ollut Android-täppäri kuuluu, vähiten tehokkaimpia. Kolmanteen kategoriaan kuuluvat pelikonsolit jäävät tyypillisesti tehoiltaan näiden kahden ääripään väliin. (Lomas 2014.) Täten testilaitteet kattavat tämän hetkiset videopelilaitteet riittävän hyvin, eikä kattavammasta laitekattauksesta olisi ollut paljoakaan hyötyä, koska opinnäytetyössä käsiteltävien optimointikeinojen periaatteet pätevät kaikissa laitteissa. Sen sijaan, mikäli laitekattausta olisi kasvatettu, olisi opinnäytetyön työtaakka siltä osin kasvanut, minkä vuoksi jotain muuta olisi todennäköisesti jouduttu jättämään pois.

PC (kuva 27):

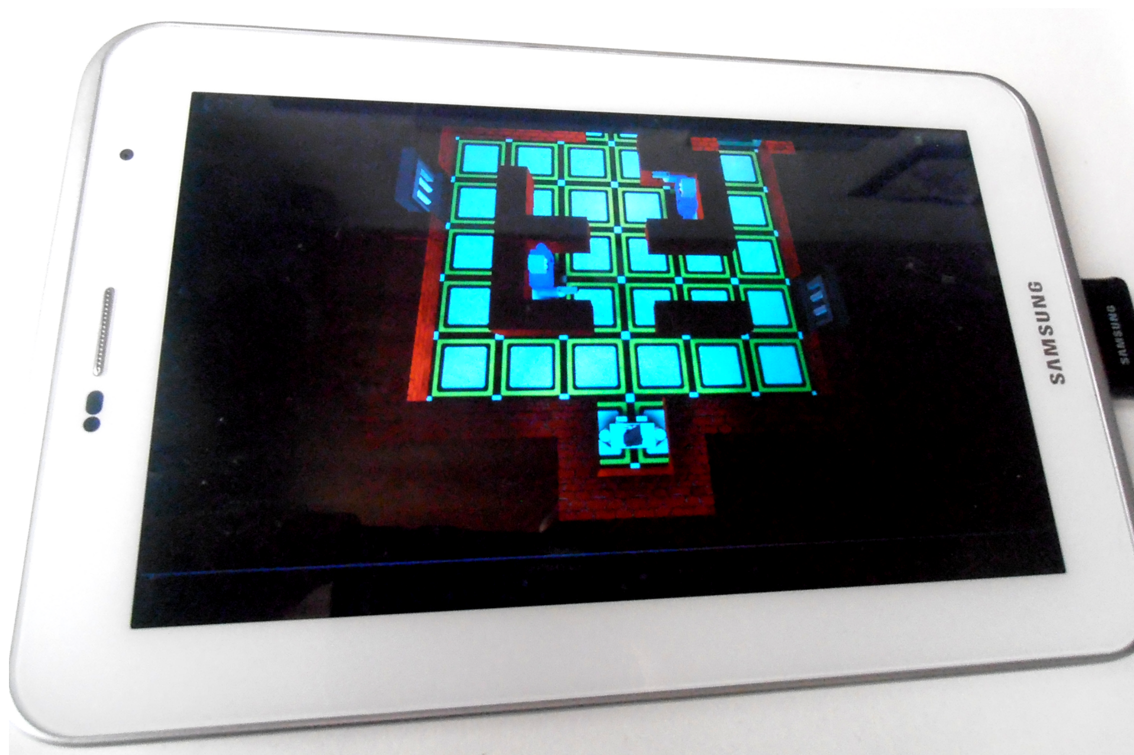
- Malli: Asus CM6870-NR0090 Desktop
- Käyttöjärjestelmä: Windows 7 Home Premium 64bit
- Suoritin: Intel CORE i7-3770 3.4GHz Quad-Core
- RAM: 8Gt
- Näytönohjain: NVIDIA GeForce GT 640
- Näytön resoluutio: 1920 x 1080
- Näytön kuvasuhde: 16:9



Kuva 27. Testilaitteena käytetty PC-tietokone ja sen näyttö.

Android-täppäri (kuva 28):

- Malli: Samsung GALAXY Tab 2 7.0
- Käyttöjärjestelmä: Android 4.0 (Ice Cream Sandwich)
- Suoritin: OMAP4430 1GHz Dual-Core
- RAM: 1Gt
- Näytön resoluutio: 1024 × 600
- Näytön kuvasuhde: 16:10



Kuva 28: Testilaitteena käytetty Android-täppäri.

8 Optimointityökalut

Optimointia varten tarvittiin myös joitain optimointityökaluja, joiden valinnasta kerrotaan tässä luvussa. Samalla kerrotaan lyhyesti miten työkaluja käytetään, mihin niitä on käytetty ja miksi. Unityn omaa profilointityökalua, Unity Profileria, ei tässä opinnäytetyössä ole käytetty, koska se on yksi Unityn Pro-versioon kuuluvista liitännäisistä (Unity Manual 2014m).

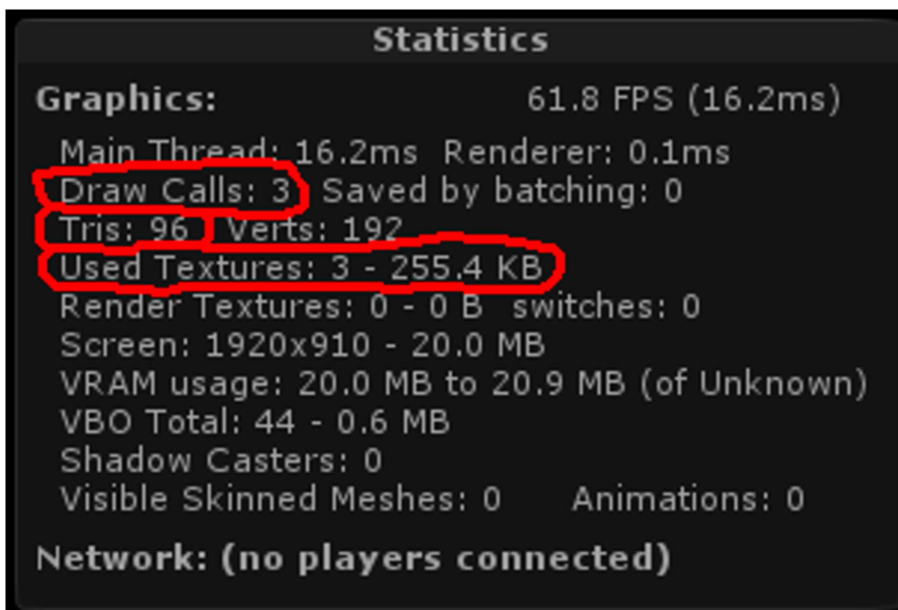
8.1 Rendering Statistics Window

Rendering Statistics Window on Unityn editoriin integroitu liitännäinen, ja se ilmoittaa grafiikoiden piirtämiseen liittyviä tietoja. Rendering Statistics Window'in käyttäminen on helppoa: se tarvitsee ainoastaan klikata auki, kun peliä ajaa Unityn editorissa. Rendering Statistics Window osaa ilmoittaa tiedot kohdealustan perusteella, joten sen avulla on mahdollista kerätä tiedot molemmille testi-

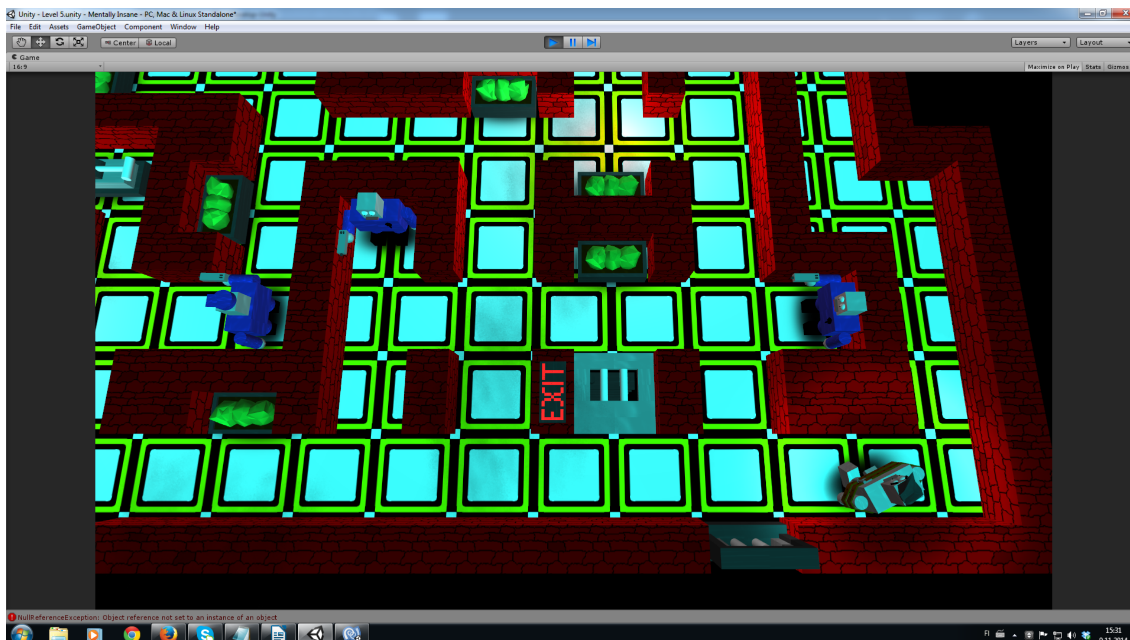
laitteille. (Unity Manual 2014n.)

Tässä opinnäytetyössä Rendering Statistics Window'ta käytettiin piirtokutsujen, piirrettävien kolmioiden ja tekstuurien viemän muistin määrän mittaamiseen (kuva 29). Ruudunpäivitysnopeuksien mittaamiseen sitä ei sen sijaan käytetty seuraavista syistä:

- Rendering Statistics Window ilmoittaa ruudunpäivitysnopeuden (joka vaihtelee jatkuvasti) ainoastaan reaaliaikaisesti, joten ruudunpäivitysnopeuden keskiarvon olisi joutunut arvioimaan epätarkasti sen sijaan, että sen olisi laskenut tarkasti tietokone, jolloin tuloksia ei olisi voitu pitää riittävän luotettavina.
- Kuten tuli todettua, Rendering Statistics Window'ta käytetään Unityn editorissa pelin ajon aikana, eikä pelin lopullisessa buildissa, minkä vuoksi sillä ei olisi pystytty mittaamaan ruudunpäivitysnopeuksia Androidilla ollenkaan.
- Pelejä ei pysty ajamaan PC:llä täydellä 1920 x 1080 -resoluutiolla, kun ne ovat Unityn editorissa, koska Unityn editorin painikkeet vievät hieman tilaa ruudulta (kuva 30).



Kuva 29. Kuvakaappaus Rendering Statistics Window'sta, jossa tiedot, joiden keräämiseen sitä on tässä opinnäytetyössä käytetty, on korostettu punaisella värillä.

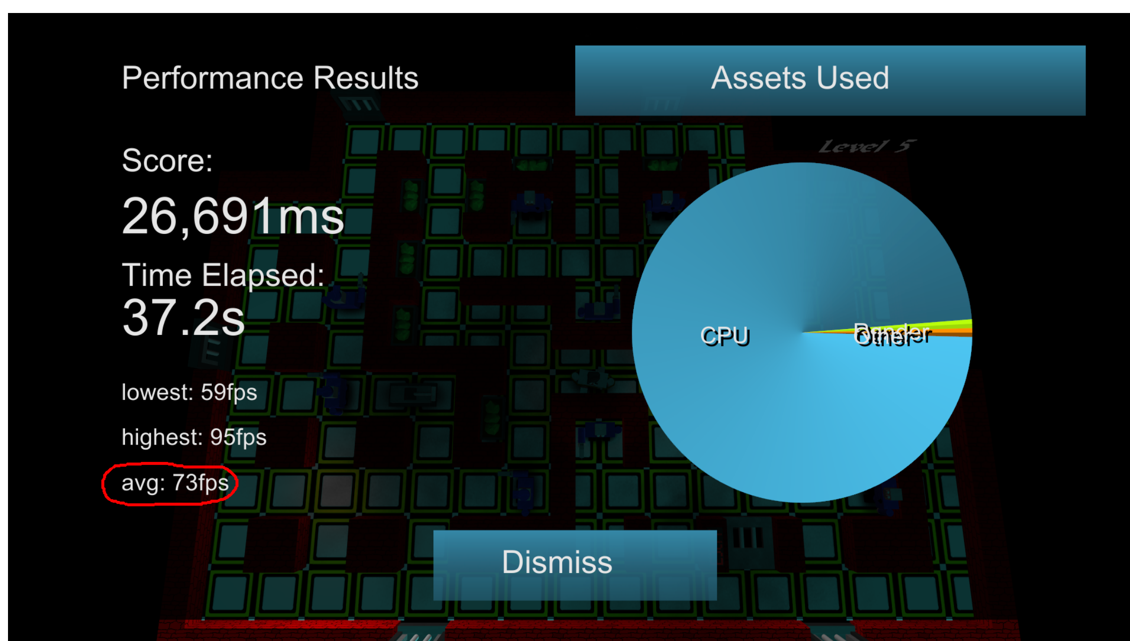


Kuva 30. Kuvakaappaus, jossa ajetaan Status: Insanea Unityn editorissa, mistä nähdään, että editorin painikkeiden viemän tilan vuoksi peliä ei pystytty ajamaan täydellä 1920 x 1080 -resoluutiolla.

8.2 FPS Graph

Kuten luvussa 8.1 tuli todettua, Rendering Statistics Window ei ollut sopiva työkalu mittaamaan pelien ruudunpäivitysnopeuksia, joten sitä tarkoitusta varten tarvittiin jokin toinen liitännäinen. Sopivaa liitännäistä etsittiin Internetistä ja sellaiseksi valikoitui FPS Graph (Dented Pixel 2014a).

Toisin kuin Rendering Statistics Window, FPS Graph osaa laskea ruudunpäivitysnopeuden keskiarvon tietyn ajanjakson ajalta (kuva 31). FPS Graphin käyttöönotto on helppoa: FPS Graph tarvitsee ainoastaan liittää tason kameraan, minkä jälkeen se on täysin valmis käytettäväksi. (Dented Pixel 2014b.) Koska FPS Graph toimii kamerassa, eikä Rendering Statistics Window'in tapaan Unityn editoriin integroituna, pystyttiin sitä käyttämään pelien buildeissa. Sen ansiosta pystyttiin mittaamaan ruudunpäivitysnopeudet PC:n lisäksi myös Androidilla ja PC:llä pystyttiin käyttämään täyttä 1920 x 1080 -resoluutiota. Käytetty FPS Graphin versio oli 0.975.



Kuva 31. Kuvakaappaus FPS Graph -liitännäisestä sen jälkeen, kun sillä on kerätty ruudunpäivityksen arvot joltain tietyltä ajanjaksolta. Ruudunpäivitysnopeuden keskiarvo, jonka keräämiseen FPS Graphia käytettiin tässä opinnäytetyössä, on korostettu punaisella värillä.

9 Optimoimisen käytänteet

Teoriaosiossa kerrotuista optimointikeinoista seuraavia testattiin tämän opinnäytetyön käytännön osiossa:

- Valaistuksen optimointi (ks. luku 4.1.1)
- Kameran asettelu (ks. luku 4.1.2)
- Tekstuurien optimointi (ks. luku 4.2.1)
- Fysiikoiden optimointi (ks. luku 5.3)

Sen sijaan seuraavat optimointikeinot jätettiin pois:

- Pelien 3D-malleihin tehtävät optimoinnit eli objektimäärän optimointi (ks. luku 4.1) sekä kolmioiden määrän optimointi (ks. luku 4.2.2) jätettiin pois, koska valmiita 3D-malleja oli tässä vaiheessa enää liian hankala optimoida, etenkin kun ne olivat jo valmiiksi ainakin osittain optimoituja.
- Tarpeettomien päivitysten välttäminen (ks. luku 5.1) jätettiin pois, koska

kummastakaan testattavassa pelistä ei löytynyt sopivaa tilannetta, jossa sitä olisi voitu soveltaa: Unityn sisäänrakennettuja funktioita oli käytetty optimaalisesti. Ainut selkeä poikkeus oli Puck Buddiesin tulokset-taso, jossa koodi ei ollut kovinkaan optimaalista (kaikki toiminnallisuus oli OnGUI-funktion sisällä), mutta tätä tilannetta varten olisi jouduttu luomaan oma erillinen testitilanteensa, koska muut testit tehtiin eri tasossa. Tämän vuoksi se ei olisi tuonut juurikaan lisäarvoa tähän opinnäytetyöhön.
- Objektien altaus (ks. luku 5.2) jätettiin pois, koska kummastakaan testattavassa pelistä ei löytynyt sopivaa tilannetta, jossa luotaisiin runsaasti uusia objekteja ja tuhottaisiin vanhoja.

9.1 Alkutilanne ja testaamisen suorittaminen

Ennen kunkin optimointikeinon testausta pelit asetettiin alkutilanteeseen, joka on tilanne, jossa mitään testattavista optimointikeinoista ei ole käytetty. Tällöin saadut tulokset kertoivat, kuinka paljon kukin optimointikeino paransi pelin suorituskykyä verrattuna alkutilanteeseen, minkä ansiosta tuloksia pystyttiin vertailemaan keskenään.

Testaaminen suoritettiin siten, että ensiksi tehtiin testit ja kerrottiin tulokset (luvut 10 ja 11), minkä jälkeen luvussa 12 tulokset arvioitiin.

9.2 Tietojen kerääminen

Optimointikeinoja testatessa oli tarpeen kerätä erilaisia tietoja talteen, jotta pystyttiin havainnoimaan paremmin miten kukin optimointikeino vaikutti pelien suorituskykyyn. Seuraavaksi kerrotaan, mitä tietoja kerättiin ja miksi:

- **Alusta**
    - PC tai Android.
    - Tämä tieto kerättiin, koska eri alustojen kesken laitteiston resurs-

seissa oli suuria eroja (ks. luku 7), joten se vaikutti suuresti myös pelien suorituskykyyn.

- **Resoluutio**
  - Yksikkö: ruudun resoluutio pikseleinä (ruudun leveys kertaa ruudun korkeus).
  - Tämä tieto kerättiin, koska käytetty ruudun resoluutio vaikutti grafiikkaprosessorin suorituskykyyn (ks. luku 4).
- **Piirtokutsut**
  - Yksikkö: piirtokutsujen määrä yhden ruudunpäivityksen aikana.
  - Tämä tieto kerättiin, koska piirtokutsut vaikuttivat prosessorin suorituskykyyn (ks. luku 4.1).
- **Kolmiot**
  - Yksikkö: piirrettävien kolmioiden määrä.
  - Tämä tieto kerättiin, koska piirrettävien kolmioiden määrä vaikutti etenkin grafiikkaprosessorin ja jonkin verran myös prosessorin suorituskykyyn (ks. luku 4.2.2).
- **Tekstuurit**
  - Yksikkö: tekstuurien viemä muistin määrä.
  - Tämä tieto kerättiin, koska tekstuurien viemä muistin määrä vaikutti grafiikkaprosessorin suorituskykyyn (ks. luku 4.2).
- **Ruudunpäivitys**
  - Yksikkö: kuvia keskimäärin sekunnissa.
  - Tämä tieto kerättiin, koska ruudunpäivitysnopeuksia tarkkailemalla saatiin selville pelin silloinen suorituskyky.

Taulukko 1 on esimerkki tietojen keräämistä varten tehdystä taulukosta. Taulukossa kerrotaan jonkin kuvitteellisen PC-pelin tilastot jonkin kuvitteellisen optimointikeinon käyttämisen jälkeen, minkä lisäksi taulukoissa on nähtävissä vertailun vuoksi myös pelin kuvitteellisen alkutilanteen tilastot. Esimerkkitaulukosta nähdään, että käytetty optimointikeino on vähentänyt piirtokutsujen määrää 1000:lla ja ruudunpäivitysnopeus on noussut viidellä kuvalla sekunnissa verrattuna alkutilanteeseen.

Taulukko 1. Esimerkkitaulukko, jollaisiin tietoja kerättiin.

ALUSTA	PC nyt	PC alkutilanteessa
RESOLUUTIO	1920 x 1080	1920 x 1080
PIIRTOKUTSUT	1000	2000
KOLMIOT	~123 000	~123 000
TEKSTUURIT	~1.1MB	~1.1MB
RUUDUNPÄIVITYS	~20 (+5)	~15

Joissain tiedoissa käytettiin tarkkojen arvojen sijaan likiarvoja, jotka on merkitty ~-merkillä. Tämä johtuu seuraavista seikoista:

- Kolmiot: Rendering Statistics Window pyöristi piirrettävien kolmioiden määrän automaattisesti.
- Tekstuurit: Rendering Statistics Window pyöristi tekstuurien viemän muistin määrän automaattisesti.
- Ruudunpäivitys: FPS Graph pyöristi ruudunpäivitysnopeuksien keskiarvot kokonaisluvuiksi automaattisesti, minkä lisäksi niiden tarkka mittaaminen oli käytännössä mahdotonta (ks. luku 9.3).

9.3 Laadun tarkkailu

Jotta kerättyjä tietoja ja niiden perusteella tehtyjä johtopäätöksiä pystyi pitämään luotettavina, oli syytä kiinnittää huomiota myös tietojen keräämismetodien laatuun. Näihin asioihin kiinnitettiin erityistä huomiota:

- Kustakin pelistä valittiin optimoitavaksi taso, joka kuormitti testilaitteita mahdollisimman paljon. Tämä tehtiin sen takia, että mikäli testilaitteet suoriutuivat eniten tehoja vaativasta tasosta hyvin, oli syytä olettaa, ettei myöskään muiden, vähemmän tehoja vaativien, tasojen kanssa tulisi ongelmia.
- Testiolosuhteista pyrittiin saamaan mahdollisimman yhdenmukaiset minimoimalla erilaisten muuttujien määrää. Tämän vuoksi kaikki tarpeettomat tietokoneohjelmat suljettiin testaamisen ajaksi, minkä lisäksi peleihin tehtiin joitain muutoksia, jotta kerättyjen tietojen arvot vakiintuivat paikoilleen

sen sijaan, että ne olisivat vaihdelleet jatkuvasti.

- Mikäli peleihin jouduttiin yhdenmukaisten testiolosuhteiden aikaansaamiseksi tekemään muutoksia, nämä muutokset pyrittiin pitämään mahdollisimman vähäisinä: joissain tapauksissa pieni satunnaisuuden lisääntyminen oli hyväksyttävää mikäli peliin olisi joutunut muuten tekemään liikaa muutoksia. Peleihin tehdyistä muutoksista on kerrottu tarkemmin ennen kunkin pelin optimoinnin aloitusta luvuissa 10 ja 11.
- Testausajat pidettiin riittävän pitkinä, koska ruudunpäivitysnopeudet vaihtelivat ajon aikana paljon[7], minkä vuoksi liian lyhyet testausajat olisivat voineet antaa vääristyneitä ruudunpäivitysnopeuksien keskiarvoja. Tämän lisäksi testit toistettiin vähintään kolme kertaa, joista laskettiin tarvittaessa ruudunpäivitysnopeuksien keskiarvojen keskiarvo, koska mitatuissa ruudunpäivitysnopeuksissa oli usein hieman virhemarginaalia[8] vaikka testausajat olivatkin pitkiä.

9.4 Laatuasetusten säätäminen

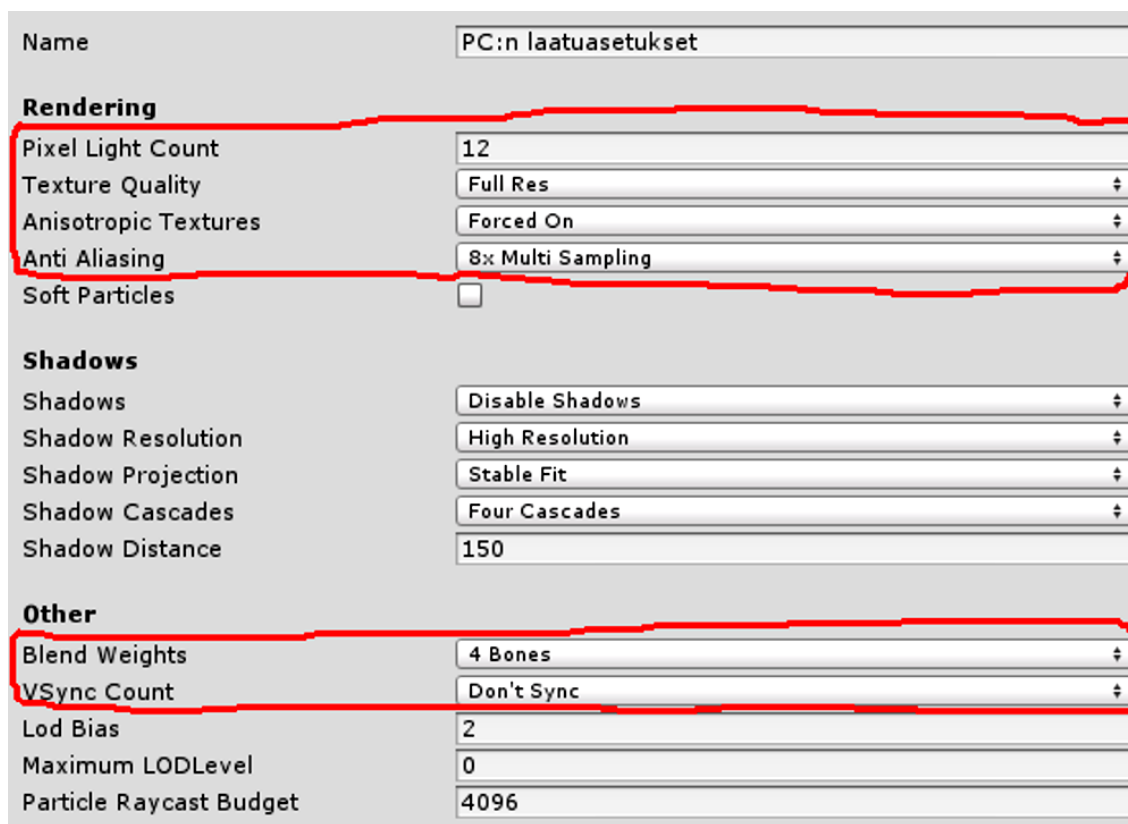
Koska testilaitteena käytetty PC-tietokone oli riittävän tehokas, pystyttiin laatuasetuksista säätämään sille parhaat asetuksien arvot (kuva 32):

- Pikselivalojen maksimimääräksi (engl. pixel light count) asetettiin 12, mikä käytännössä tarkoitti sitä, että kaikissa valonlähteissä käytettiin pikselikohtaista valaistusta, koska peleissä oli valonlähteitä vähemmän kuin 12.
- Tekstuurien laaduksi (engl. texture quality) asetettiin täysi resoluutio.
- Anisotrooppiset tekstuurit olivat käytössä.

7 Korkeimman ja matalimman ruudunpäivitysnopeuden arvon erotus oli PC:llä useamman kymmenen ruudunpäivityksen luokkaa, kun taas Andoidilla se oli noin kymmenen ruudunpäivityksen luokkaa.

8 PC:llä ruudunpäivitysnopeuden keskiarvot vaihtelivat yhdellä tai maksimissaan kahdella kuvalla yli tai ali sen arvon, joka kirjattiin taulukkoon. Eli mikäli taulukkoon kirjattiin esimerkiksi arvo 180 kuvaa sekunnissa, ruudunpäivitysnopeuksien keskiarvot vaihtelivat tällöin 178–182 kuvan välillä. Androidilla sen sijaan vaihtelua oli vähemmän: Androidilla saatiin usein jokaisella kolmella testillä täysin sama ruudunpäivitysnopeus.

- Reunojen pehmennyksessä käytettiin parasta laatua eli kahdeksankertaista pehmennystä.
- Painopisteiden koostamisessa käytettiin parasta laatua eli neljää luuta.
- Ruudunpäivitysnopeuden synkronointi (engl. vsync count) otettiin pois käytöstä, koska muutoin Unity ei olisi antanut ruudunpäivitysnopeuden mennä yli 60:n kuvan sekunnissa. Tällöin ruudunpäivitysnopeuksien vertailu olisi ollut mahdotonta aina, kun ruudunpäivitysnopeus olisi mennyt sen yli.



Kuva 32. Kuvakaappaus PC:n laatuasetuksista, joista merkitykselliset asetukset on korostettu punaisella värillä.

Kun Status: Insanen alkutilannetta testattiin parhailla laatuasetuksien arvoilla testilaitteena käytetyllä Android-täppärillä, ruudunpäivitysnopeus oli ainoastaan kaksi kuvaa sekunnissa, minkä vuoksi peliä oli todella epämukava pelata. Tämän takia Androidilla käytettiin PC:stä poiketen huonoimpia laatuasetusten arvoja. Unityn manuaalissakin (2014o) kerrotaan, ettei mobiililaitteille usein kannattaisi käyttää parhaita laatuasetuksien arvoja, joten alhaisempien arvojen käyttäminen oli täysin perusteltua. Androidilla käytettiin seuraavanlaisia laatu-

asetuksien arvoja (kuva 33):

- Pikselivalojen maksimimääräksi asetettiin 0 eli pikselikohtaista valaistusta ei käytetty ollenkaan.
- Tekstuurien laaduksi asetettiin huonoin mahdollinen, joka oli 1/8-resoluutio.
- Anisotrooppiset tekstuurit eivät olleet käytössä.
- Reunojen pehmennys ei ollut käytössä.
- Painopisteiden koostamisessa käytettiin huonointa laatua eli yhtä luuta.
- Ruudunpäivitysnopeuden synkronointi ei ollut käytössä.

Name	Androidin laatuasetukset
Rendering	
Pixel Light Count	0
Texture Quality	Eighth Res
Anisotropic Textures	Disabled
Anti Aliasing	Disabled
Soft Particles	☐
Shadows	
Shadows	Disable Shadows
Shadow Resolution	High Resolution
Shadow Projection	Stable Fit
Shadow Cascades	Four Cascades
Shadow Distance	150
Other	
Blend Weights	1 Bone
VSync Count	Don't Sync
Lod Bias	0
Maximum LODLevel	0
Particle Raycast Budget	0

Kuva 33. Kuvakaappaus Androidin laatuasetuksista, joista merkitykselliset asetukset on korostettu punaisella värillä.

Laatuasetuksissa oli lisäksi asetuksia, joista seuraavilla ei ollut mitään vaikutusta testattaviin peleihin:

- Pehmeiden partikkelien (engl. soft particles) piirtämiseksi olisi pitänyt käyttää jaksotetusti valaistua piirtämispolkua (Unity Manual 2014o). Toinen vaihtoehto olisi ollut skriptaaminen (Unity Manual 2014o), mutta se

olisi ollut tämän opinnäytetyön puitteissa liian työlästä tehdä.

- Varjojen laatuun vaikuttavilla asetuksilla ei ollut merkitystä, koska dynaamiset varjot kuuluvat Unityn Pro-version ominaisuuksiin (Unity Manual 2014p).
- Yksityiskohtien tasoihin (engl. level of detail eli LOD) -liittyvillä asetuksilla ei ollut merkitystä, koska yksityiskohtien tasojen säätäminen kuuluu Unityn Pro-version ominaisuuksiin (Unity Manual 2014q).
- Partikkelien säteidenluontibudjetilla (engl. particle raycast budget) ei ollut merkitystä, koska kummankaan testipelin partikkelisysteemeissä ei ollut käytetty Collision Module -moduulia, jolla partikkelit olisi saatu kimpoilemaan osuessaan kiinteisiin objekteihin (Unity Manual 2014r).

10 Status: Insanen optimointi

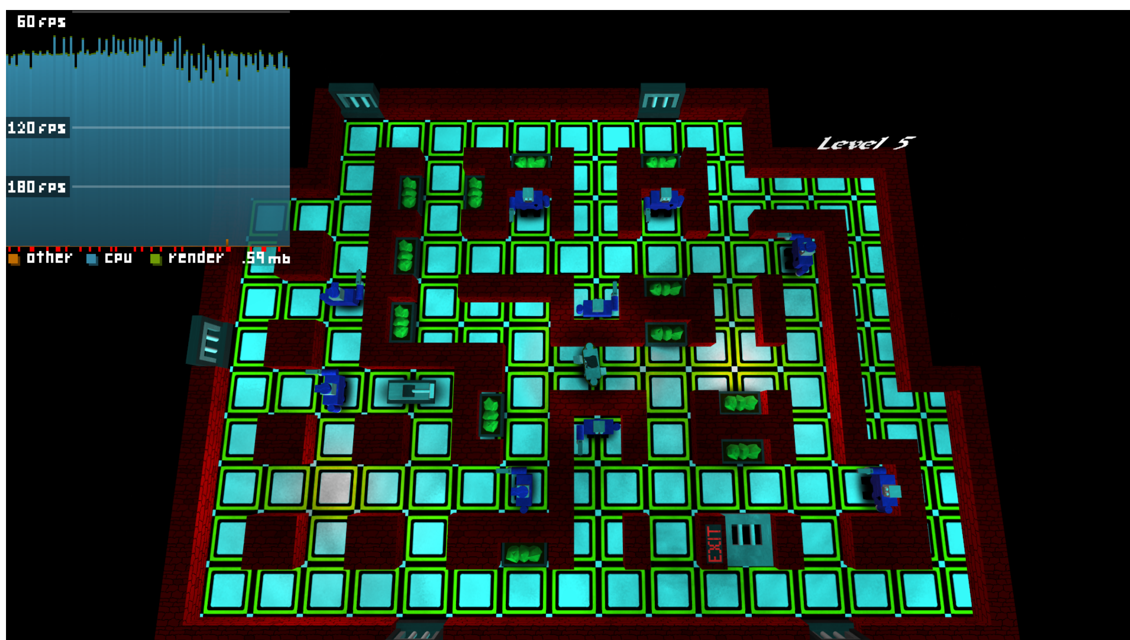
Status: Insanesta valittiin optimoitavaksi pelin viides taso, koska se oli tasoista kaikkein kehittynein: objektien määrä oli tässä tasossa kaikkein suurin ja tasossa oli eniten kääntyviä vartijoita. Testiolosuhteiden yhdenmukaisuuden aikaansaamiseksi ei tarvinnut tehdä juuri mitään, koska kerättävien tietojen arvot vakiintuivat paikoilleen, kun pelaajan ohjaama pelihahmokin pysyi paikallaan. Tämän vuoksi kaikkien testien aikana pelaajan ohjaama pelihahmo seisoi keskellä tasoa täysin liikkumattomana.

Testiolosuhteisiin tosin lisäsi hieman satunnaisuutta partikkelisysteemi, joka loi tasoon savua. Savun määrä pysyi kuitenkin suunnilleen vakiona, minkä vuoksi kerätyt arvotkin vakiintuivat paikoilleen, eikä tätä partikkelisysteemiä tämän vuoksi kannattanut ottaa pois testaamisen ajaksi.

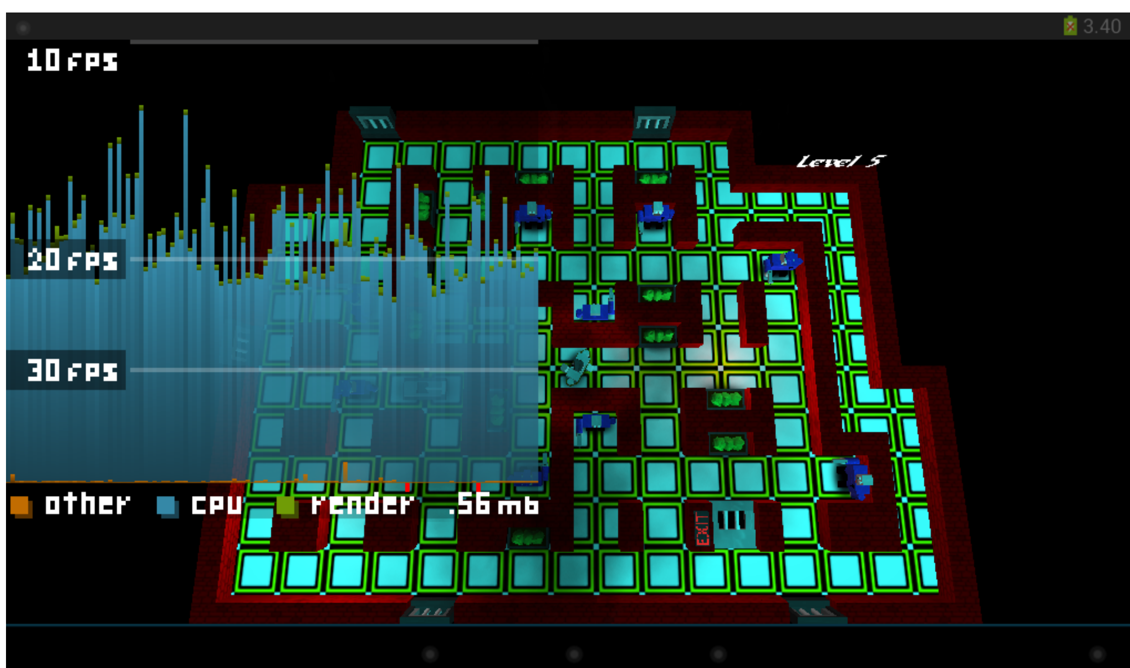
10.1 Alkutilanne

Status: Insanen alkutilanteessa (kuvat 34 ja 35) ei ollut käytetty mitään optimointikeinoja:

- **Valaistus:**
    - Yksi Directional Light -komponentti, jonka dynaamisen valonlähteen tyypiksi oli asetettu automaattinen.
    - Kaksi Point Light -komponenttia, joiden dynaamisten valonlähteiden tyypiksi oli asetettu automaattinen.
    - Valokartoitusta ollut käytetty.
- **Kamera:**
    - Kamera oli asetettu siten, että kaikki pelialueella olevat objektit jouduttiin piirtämään kerralla.
- **Tekstuurit:**
    - Tekstuurit olivat pakkaamattomia: 3D-malleissa käytetyt kehittyneemmät tekstuurit olivat 32-bittisiä RGBA-tyyppisiä tekstuureita ja muut tekstuurit truecolor-tyyppisiä tekstuureita.
    - Mip-kartoitusta ei ollut käytetty.
- **Fysiikat:**
    - Muunnetun aika-askeleen arvoksi oli asetettu Unityn antama oletusarvo eli 0.02, mikä tarkoitti sitä, että fysiikoita päivitettiin 50 kertaa sekunnissa.
    - Kaikki käytetyt törmäyttimet olivat laatikkotörmäyttimiä, minkä lisäksi staattisille objekteille oli käytetty staattisia törmäyttimiä.



Kuva 34. Kuvakaappaus Status: Insanen PC-version viidennestä tasosta alkutilanteessa, minkä ruudunpäivitysnopeuksia FPS Graph -liitännäinen analysoi kuvan vasemmassa yläkulmassa.



Kuva 35. Kuvakaappaus Status: Insanen Android-version viidennestä tasosta alkuatilanteessa, minkä ruudunpäivitysnopeuksia FPS Graph -liitännäinen analysoi kuvan vasemmassa yläkulmassa.

Taulukosta 2 nähdään, että piirtokutsujen, kolmioiden ja tekstuurien viemän muistin määrissä oli merkittäviä eroja eri versioiden välillä alkutilanteessa: PC:llä piirtokutsujen sekä kolmioiden määrä oli lähes kolminkertainen ja teks-

tuurien viemä muistin määrä yli 19-kertainen verrattuna Androidiin. Lisäksi PC:n maksimiresoluutio oli merkittävästi Androidin maksimiresoluutiota suurempi: PC:llä ruudulle piirrettävien pikselien yhteenlaskettu määrä oli yli kolminkertainen verrattuna Androidiin[9]. Tästä kaikesta huolimatta ruudunpäivitysnopeus oli PC:llä yli neljä kertaa suurempi kuin Androidilla.

Taulukko 2. Status: Insanen tilastot alkutilanteessa.

ALUSTA	**PC alkutilanteessa**	**Android alkutilanteessa**
RESOLUUTIO	1920 x 1080	1024 x 600
PIIRTOKUTSUT	1241	425
KOLMIOT	~100 100	~34 500
TEKSTUURIT	~2.7MB	~140.8KB
RUUDUNPÄIVITYS	~75	~17

10.2 Grafiikoiden suorituskyvyn pullonkaula

Kuten luvussa 4 kerrottiin, ennen grafiikoiden optimoinnin aloittamista tuli selvittää, oliko grafiikoiden suorituskyvyn pullonkaulana prosessori vai grafiikkaprosessori. Keinoksi pullonkaulan paikantamiseksi kerrottiin, että piti testata paransiko resoluution alentaminen ruudunpäivitysnopeutta. Androidilla tätä ei pystytty kuitenkaan testaamaan, koska Unity määritti Androidin resoluution testilaitteen perusteella automaattisesti, mutta PC:llä tätä ongelmaa ei onneksi ollut (Unity Manual 2014s).

Taulukko 3 kertoo tilastot PC:llä, kun sen resoluutio oli laskettu alkutilanteen 1920 x 1080 pikselistä 1280 x 720 pikseliin. Kuten taulukosta nähdään, resoluution laskeminen ei muuttanut tilastoista muita arvoja kuin ruudunpäivitysnopeuden, joka parantui 59:llä kuvalla sekunnissa eli yhteensä 79:llä %:lla verrattuna alkutilanteeseen. Tästä pystyi tekemään sen johtopäätöksen, että grafiikoiden

9 PC:llä resoluutio oli 1920 x 1080 pikseliä, jolloin pikseleitä piirrettiin ruudulle yhteensä 2073600 kappaletta (1920*1080=2073600). Androidilla resoluutio oli sen sijaan 1024 x 600 pikseliä, jolloin pikseleitä piirrettiin ruudulle yhteensä 614400 kappaletta (1024*600=614400).

suorituskyvyn pullonkaulana oli todennäköisesti grafiikkaprosessori.

Taulukko 3. Status: Insanen tilastot resoluution laskemisen jälkeen.

ALUSTA	PC nyt	PC alkutilanteessa
RESOLUUTIO	1280 x 720	1920 x 1080
PIIRTOKUTSUT	1241	1241
KOLMIOT	~100 100	~100 100
TEKSTUURIT	~2.7MB	~2.7MB
RUUDUNPÄIVITYS	~134 (+59)	~75

10.3 Valaistuksen optimointi

Tasossa oli alkutilanteessa yksi Directional Light -komponentti ja kaksi Point Light -komponenttia, joiden valonlähteiden tyypiksi oli asetettu automaattinen. Alkutilanteessa ei ollut myöskään käytetty valokartoitusta. Directional Light -komponenttia ei kuitenkaan voitu optimoida mitenkään, mikä johtui käytetystä piirtämispolusta (Unity Manual 2014t, ks. luku 4.1.1). Se ei kuitenkaan haitannut, koska valaistuksen pullonkaula oli kahdessa käytetyssä Point Light -komponentissa, sillä edes Directional Light -komponentin ottaminen pois tasosta ei juurikaan parantanut ruudunpäivitysnopeutta.

Ensiksi testattiin, että miten pikselikohtaisen valaistuksen muuttaminen verteksivalaistukseksi vaikutti pelin suorituskykyyn. Taulukosta 4 nähdään, että PC:llä se laski piirtokutsujen ja piirrettävien kolmioiden määrän samaan kuin Androidilla alkutilanteessa, mutta tekstuurien viemässä muistin määrässä ei tapahtunut lainkaan muutosta. Ruudunpäivitysnopeus nousi tällöin PC:llä yli kaksinkertaiseksi, kun taas Androidilla ei tässä vaiheessa tapahtunut mitään muutoksia.

Taulukko 4. Status: Insanen tilastot, kun käytössä oli verteksivalaistus, muttei valokartoitusta.

ALUSTA	**PC nyt**	**PC alkutilanteessa**	**Android nyt (ja alkutilanteessa)**
DYNAAMISEN VALAISTUKSEN TYYPPI	verteksivalaistus	pikselikohtainen valaistus	verteksivalaistus
VALOKARTOI-TUS	ei käytetty	ei käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600
PIIRTOKUTSUT	425	1241	425
KOLMIOT	~34 500	~100 100	~34 500
TEKSTUURIT	~2.7MB	~2.7MB	~140.8KB
RUUDUNPÄIVI-TYS	~164 (+89)	~75	~17

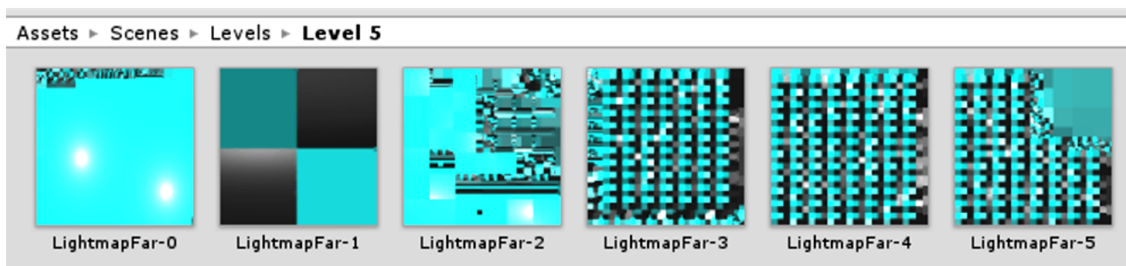
Seuraavaksi tehtiin valokartoitus staattisille objekteille, joita olivat paikallaan seisovat vartijat, lattia, seinät, kasvit, ikkunat, kyltti, jossa lukee ”EXIT” ja Level 5 -teksti. Sen sijaan dynaamiset objektit, joita olivat pelaajan ohjaama pelihahmo, kääntyvät vartijat, vedettävä vipu ja aukeava luukku, käyttivät yhä edelleen dynaamista valaistusta.

Valokartoituksessa käytettiin yksinkertaisia valokarttoja sekä Unityn antamaa oletusresoluutiota, joka oli 50 pikseliä maailman yksikköä kohti. Erilaisia valokarttoihin lisättäviä tehosteita, kuten ympäristön okkluusiota (engl. ambient oclusion), ei käytetty, vaan valokartoituksen vaikutus pyrittiin pitämään mahdollisimman neutraalina. Sen ansiosta tuloksia arvioidessa pystyttiin paremmin arvioimaan valokartoituksen vaikutusta valaistuksen laatuun.

Yllä kuvatuilla asetuksilla valokartoitusliitännäinen generoi tason nimen mukaisesti Level 5 -nimisen kansion sekä sinne kuusi[10] valokartta-tekstuuria, joista

10 Valokartoitusta testattiin myös resoluutiolla 10, jolloin valokarttoja generoitiin ainoastaan yksi kappale, mutta grafiikoiden laatu heikkeni selkeästi. Lisäksi testattiin resoluutiota 100, jolloin valokarttoja generoitiin peräti 20 kappaletta, mutta grafiikoiden laadussa ei näkynyt muutosta resoluutioon 50 verrattuna, minkä vuoksi resoluutio 50 todettiin riittävän hyväksi.

jokaisen resoluutio oli 1024 x 1024 pikseliä (kuva 36).



Kuva 36. Kuvakaappaus Level 5 -kansion sisällöstä.

Taulukosta 5 nähdään, että PC:llä valokartoitus miltei nelinkertaisti tekstuurien viemän muistin määrän, mutta piirtokutsujen ja piirrettävien kolmioiden määrät laskivat melko paljon. Samalla ruudunpäivitysnopeus parantui noin 11:llä kuvalla sekunnissa eli noin 15:llä %:lla verrattuna alkutilanteeseen. Androidilla tekstuurien viemä muistin määrä sen sijaan nousi PC:tä huomattavasti vähemmän, kun taas piirtokutsujen ja piirrettävien kolmioiden määrä pysyivät ennallaan. Siitä huolimatta ruudunpäivitysnopeus parantui kuitenkin neljällä kuvalla sekunnissa eli noin 24:llä %:lla verrattuna alkutilanteeseen.

Taulukko 5. Status: Insanen tilastot, kun käytössä oli pikselikohtainen valaistus ja valokartoitus.

ALUSTA	**PC nyt**	**PC alkutilanteessa**	**Android nyt**	**Android alkutilanteessa**
DYNAAMISEN VALAISTUKSEN TYYPPI	pikselikohtainen valaistus	pikselikohtainen valaistus	verteksivalaistus	verteksivalaistus
VALOKARTOITUS	käytetty	ei käytetty	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	539	1241	425	425
KOLMIOT	~57 200	~100 100	~34 500	~34 500
TEKSTUURIT	~10.7MB	~2.7MB	~204.7KB	~140.8KB

RUUDUNPÄIVITYS	~86 (+11)	~75	~21 (+4)	~17

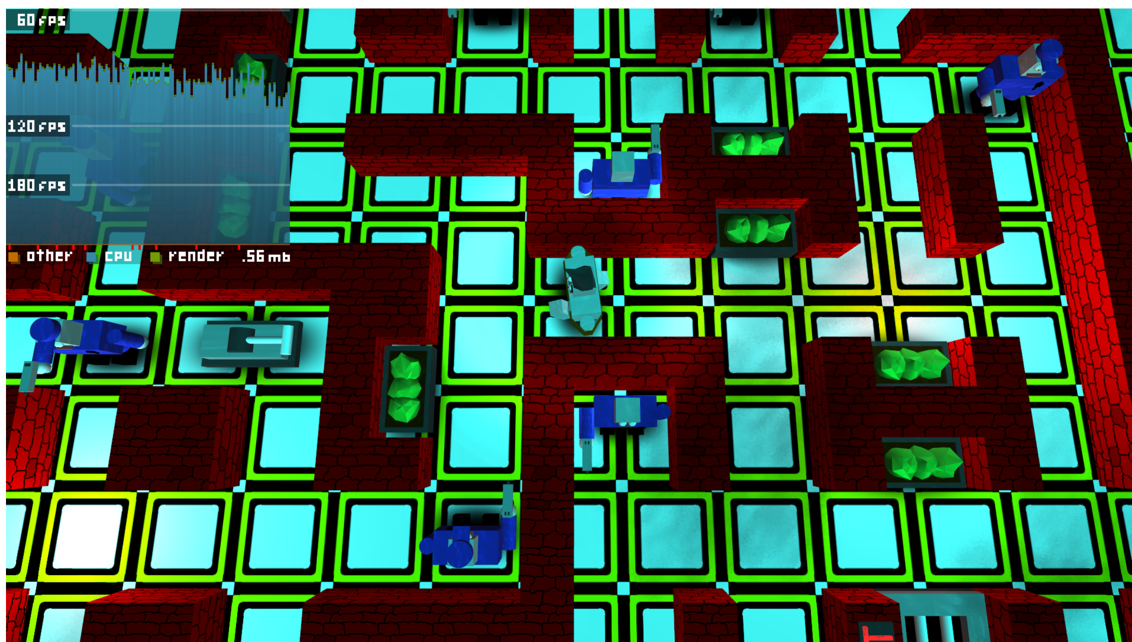
Lopuksi testattiin molempia valaistuksen optimointikeinoja yhdessä. Taulukosta 6 nähdään, että PC:llä piirtokutsujen ja piirrettävien kolmioiden määrä oli sama kuin ne olivat, kun käytettiin verteksivalaistusta ilman valokartoitusta. Piirrettävien tekstuurien määrä oli sen sijaan sama, kun käytettiin pikselikohtaista valaistusta valokartoituksen kanssa. Ruudunpäivitysnopeus nousi yli kaksinkertaiseksi verrattuna alkutilanteeseen, mutta se jäi silti alle sen, mitä se oli, kun verteksivalaistusta käytettiin ilman valokartoitusta. Androidilla kaikki tilastot olivat tismalleen samat kuin ne olivat, kun käytettiin pikselikohtaista valaistusta valokartoituksen kanssa.

Taulukko 6. Status: Insanen tilastot, kun sekä verteksivalaistus että valokartoitus olivat käytössä.

ALUSTA	PC nyt	PC alkutilanteessa	Android nyt	Android alkutilanteessa
DYNAAMISEN VALAISTUKSEN TYYPPI	verteksivalaistus	pikselikohtainen valaistus	verteksivalaistus	verteksivalaistus
VALOKARTOITUS	käytetty	ei käytetty	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	425	1241	425	425
KOLMIOT	~34 500	~100 100	~34 500	~34 500
TEKSTUURIT	~10.7MB	~2.7MB	~204.7KB	~140.8KB
RUUDUNPÄIVITYS	~155 (+80)	~75	~21 (+4)	~17

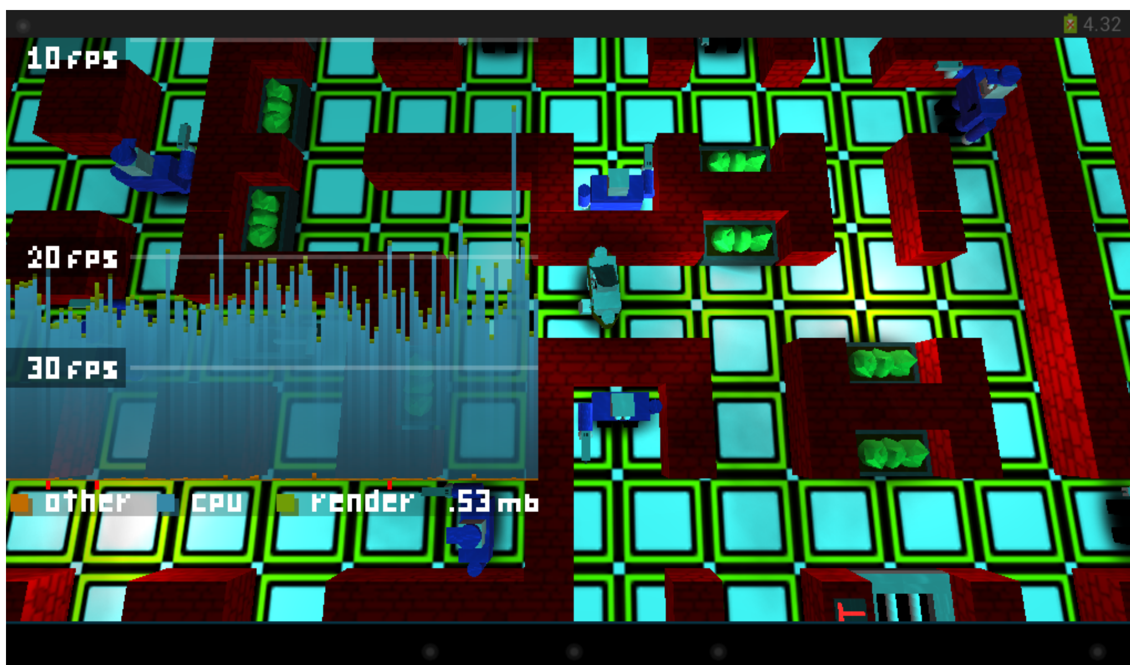
10.4 Kameran asettelu

Tason kamera oli alkutilanteessa asetettu siten, että kaikki pelialueen objektit jouduttiin piirtämään kerralla. Jotta saatiin selville kameran etäisyyden vaikutus pelin suorituskykyyn, kamera vietiin mahdollisimman lähelle pelialuetta ilman, että se haittasi liiaksi pelaamista[11] (kuva 37 ja kuva 38).



Kuva 37. Kuvakaappaus Status: Insanen PC-version viidennestä tasosta, kun kamera oli viety lähemmäksi pelialuetta.

11 Tason alkutilanteessa kameran korkeus, eli tässä tapauksessa Y-akselin arvo, oli 24 ja kameran asettelun jälkeen 12. Tason lattian Y-akselin arvo oli nolla.



Kuva 38. Kuvakaappaus Status: Insanen Android-version viidennestä tasosta kun kamera oli viety lähemmäksi pelialuetta.

Taulukosta 7 nähdään, että kameran vieminen lähemmäksi pelialuetta laski piirtokutsujen, piirrettävien kolmioiden sekä tekstuurien viemää muistin määrää jonkin verran molemmilla alustoilla. Tällöin ruudunpäivitysnopeus parantui PC:llä noin 11:llä kuvalla sekunnissa eli noin 15:llä %:lla verrattuna alkutilanteeseen. Androidilla ruudunpäivitysnopeus sen sijaan parani noin kuudella kuvalla sekunnissa eli noin 35:llä %:lla verrattuna alkutilanteeseen.

Taulukko 7. Status: Insanen tilastot kameran asettelun jälkeen.

ALUSTA	PC nyt	PC alkutilanteessa	Android nyt	Android alkutilanteessa
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	776	1241	225	415
KOLMIOT	~79 300	~100 100	~27 400	~34 500
TEKSTUURIT	~2.6MB	~2.7MB	~76.8KB	~140.8KB
RUUDUNPÄIVITYS	~86 (+11)	~75	~23 (+6)	~17

10.5 Tekstuurien optimointi

Pelin tekstuurit olivat alkutilanteessa pakkaamattomia: 3D-malleissa käytetyt kehittyneemmät tekstuurit olivat 32-bittisiä RGBA-tyyppisiä tekstuureita ja muut tekstuurit truecolor-tyyppisiä tekstuureita. 3D-mallien tekstuureissa ei ollut myöskään käytetty Mip-kartoitusta. Aluksi kokeiltiin, miten paljon tekstuurien pakkaaminen paransi ruudunpäivitystä: tekstuurit pakattiin käyttämällä kehittyneempiin tekstuureihin automaattista pakkausta (engl. automatic compressed), joka valitsi tekstuureille sopivimman vaihtoehdon automaattisesti useiden vaihtoehtojen joukosta ja muihin tekstuureihin tavallista pakkausta, joka oli niille ainoa pakkausvaihtoehto.

Taulukosta 8 nähdään, että tekstuurien pakkaaminen laski PC:llä tekstuurien viemää muistin määrää noin 74:llä %:lla verrattuna alkutilanteeseen. Ruudunpäivitysnopeus nousi tällöin ainoastaan neljällä kuvalla sekunnissa eli noin viidellä prosentilla. Androidilla tekstuurien pakkaaminen laski tekstuurien viemää muistin määrää noin 37:llä %:lla verrattuna alkutilanteeseen. Ruudunpäivitysnopeus nousi tällöin noin yhdellä kuvalla sekunnissa eli noin kuudella prosentilla.

Taulukko 8. Status: Insanen tilastot, kun käytössä oli tekstuurien pakkaaminen, muttei Mip-kartoitusta.

ALUSTA	**PC nyt**	**PC alkutilanteessa**	**Android nyt**	**Android alkutilanteessa**
TEKSTUURIEN PAKKAUS	käytetty	ei käytetty	käytetty	ei käytetty
MIP-KARTOITUS	ei käytetty	ei käytetty	ei käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	1241	1241	415	415
KOLMIOT	~100 100	~100 100	~34 500	~34 500
TEKSTUURIT	~0.7MB	~2.7MB	~89.4KB	~140.8KB
RUUDUNPÄI-	~79 (+4)	~75	~18 (+1)	~17

VITYS				

Seuraavaksi testattiin Mip-kartoitusta käyttämällä Unityn oletusasetuksia. Taulukosta 9 nähdään, että Mip-kartoituksen vaikutus oli vähäinen: PC:llä kaikki tilastot ruudunpäivitysnopeutta myöten olivat täysin samat kuin alkutilanteessa. Androidilla sen sijaan tekstuurien käyttämässä muistin määrässä oli nähtävissä pieni ero alkutilanteeseen, mutta myöskään Androidilla ruudunpäivityksessä ei tapahtunut muutoksia.

Taulukko 9. Status: Insanen tilastot, kun tekstuurien pakkaaminen ei ollut käytössä, mutta Mip-kartoitus oli käytössä.

ALUSTA	**PC nyt**	**PC alkutilanteessa**	**Android nyt**	**Android alkutilanteessa**
TEKSTUURIEN PAKKAUS	ei käytetty	ei käytetty	ei käytetty	ei käytetty
MIP-KARTOITUS	käytetty	ei käytetty	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	1241	1241	415	415
KOLMIOT	~100 100	~100 100	~34 500	~34 500
TEKSTUURIT	~2.7MB	~2.7MB	~109.5KB	~140.8KB
RUUDUNPÄIVITYS	~75	~75	~17	~17

Lopuksi testattiin molempia tekstuurien optimointikeinoja yhdessä. Taulukosta 10 nähdään, että PC:llä tilastot olivat täysin samat kuin ne olivat, kun käytettiin tekstuurien pakkaamista, muttei Mip-kartoitusta. Androidilla tekstuurien viemä muistin määrä oli hieman alhaisempi kuin se oli, kun käytettiin tekstuurien pakkaamista, muttei Mip-kartoitusta, mutta ruudunpäivitysnopeudessa ei ollut muutosta siihen tilanteeseen verrattuna.

Taulukko 10. Status: Insanen tilastot, kun sekä tekstuurien pakkaaminen että Mip-kartoitus olivat käytössä.

ALUSTA	PC nyt	PC alkutilanteessa	Android nyt	Android alkutilanteessa
TEKSTUURIEN PAKKAUS	käytetty	ei käytetty	käytetty	ei käytetty
MIP-KARTOITUS	käytetty	ei käytetty	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	1241	1241	415	415
KOLMIOT	~100 100	~100 100	~34 500	~34 500
TEKSTUURIT	~0.7MB	~2.7MB	~85.5KB	~140.8KB
RUUDUNPÄIVITYS	~79 (+4)	~75	~18 (+1)	~17

10.6 Fysiikoiden optimointi

Pelin alkutilanteessa muunnetun aika-askeleen arvoksi oli asetettu Unityn antama oletusarvo eli 0.02, mikä tarkoitti sitä, että fysiikoita päivitettiin 50 kertaa sekunnissa. Aika-askeleen arvo asetettiin arvoon 0.1, jolloin fysiikoita päivitettiin ainoastaan 10 kertaa sekunnissa. Tällä ei kuitenkaan ollut mitään näkyvää vaikutusta ruudunpäivitykseen, minkä lisäksi fysiikoiden alhainen päivitysten määrä mahdollisti seinien läpi kävelemisen, mikä oli varsin kriittinen bugi pelin pelaamisen kannalta. Tähän bugiin ei auttanut edes pelaajan ohjaaman pelihahmon jäykän kappaleen törmäysten tunnistamisen asettaminen jatkuvaksi.

Pelissä oli käytetty ainoastaan laatikkotörmäyttimiä, koska kaikki pelihahmot ja muut objektit olivat todella laatikkomaisia, minkä lisäksi staattisissa objekteissa käytettiin staattisia törmäyttimiä. Tämän optimaalisemmin törmäyttimiä, ei voitu käyttää, joten törmäyttimien optimointia ei voitu tässä pelissä soveltaa.

11 Puck Buddiesin optimointi

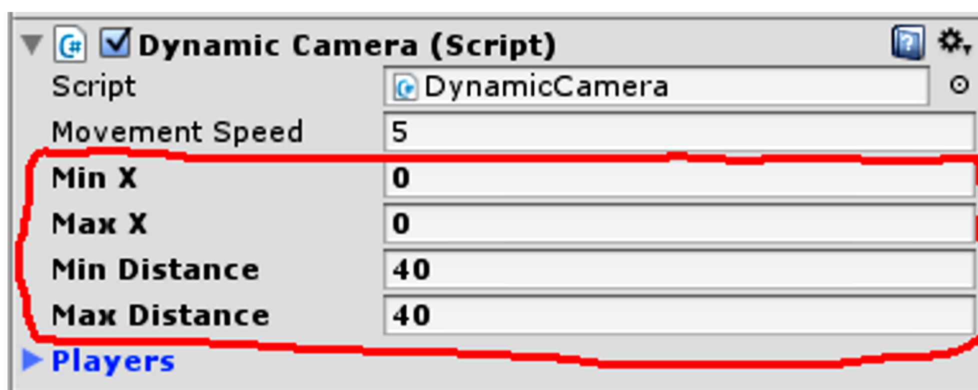
Puck Buddiesissa oli ainoastaan yksi pelitilanne-taso, mutta tason kaukalo, joita oli yhteensä kolme erilaista, oli mahdollista vaihtaa halutessaan. Optimoitavaksi valittiin kaukalo, jossa oli paikallaan pyöriviä tukkeja, koska kyseinen kaukalo oli pelin kaukaloista kaikkein kehittynein, sillä kahdessa muussa kaukalossa ei ollut mitään pyörivien tukkien kaltaisia erikoisuuksia.

Jotta testiolosuhteet olisivat olleet mahdollisimman yhdenmukaiset, täytyi pelin dynaaminen kamera lukita paikoilleen. Mikäli niin ei olisi tehty, tuloksia ei olisi voitu pitää vertailukelpoisina, koska jatkuvasti paikkaa vaihtava dynaaminen kamera (kuva 39 ja kuva 40) teki kerättyjen tietojen arvoista epätarkkoja.

Kuva 39. Kuvakaappaus Puck Buddiesista, jolloin dynaaminen kamera siirtyi lähelle pelialuetta.

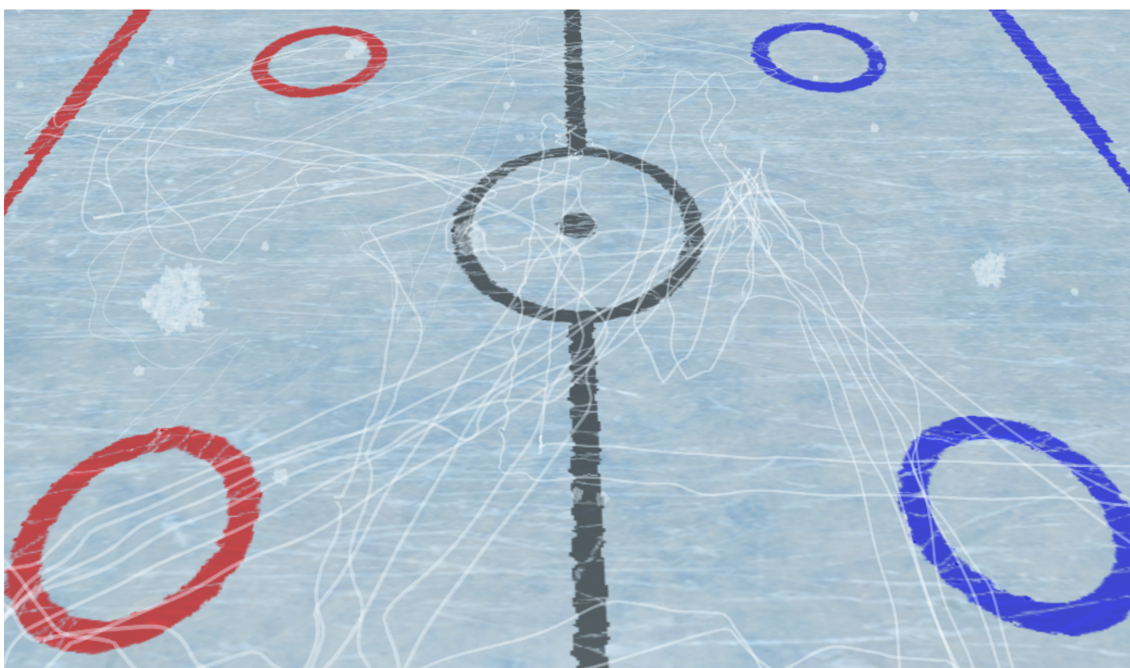
Kuva 40. Kuvakaappaus Puck Buddiesista, joilloin dynaaminen kamera siirtyi kauas pelialueesta.

Dynaamisen kameran lukitsemiseksi ei onneksi tarvinnut ottaa dynaamisen kameran skriptiä pois käytöstä, vaan ainut asia mitä tarvitsi tehdä, oli vaihtaa muuttujien arvot, jotka määrittelevät dynaamisen kameran liikkeet: muuttujien arvot vaihdettiin sellaisiksi, että dynaaminen kamera vaihtoi paikkaa identtisten arvojen välillä (kuva 41). Tämä oli hyvä asia, koska dynaaminen kamera oli tärkeä osa lopullista peliä ja tavoitteena oli pitää peli mahdollisimman paljon alkuperäisen kaltaisena.

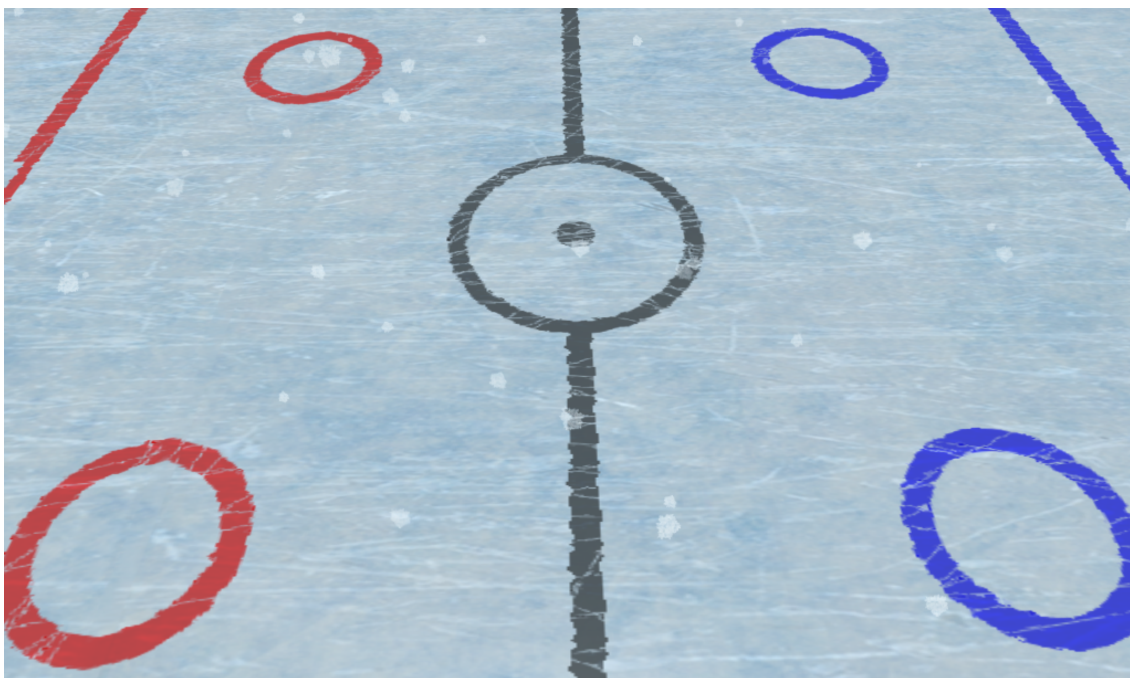


Kuva 41. Kuvakaappaus dynaamisen kameran asetuksista, joista on korostettu punaisella värillä muunnetut muuttujien arvot.

Pelkkä dynaamisen kameran lukitseminen paikoilleen ei kuitenkaan riittänyt vakiinnuttamaan kerättyjen tietojen arvoja paikoilleen, sillä piirrettävien kolmioiden määrä kasvoi jostain syystä pelin edetessä tavalla, joka vaikutti sattumanvaraiselta. Ongelman lähteeksi paljastui Trail Renderer -komponentti, joka piirsi jään pintaan pelaajien luistinten aiheuttamat urat (kuva 42), joiden määrä luonnollisesti kasvoi pelin edetessä. Kun komponentti otettiin pois käytöstä (kuva 43), kolmioiden määrä vakiintui paikoilleen, mikä tosin antoi pelille hiukan suorituskyvyllistä etua. Sen käytöstä pois ottamisesta oli kuitenkin enemmän hyötyä kuin haittaa, koska kerätyt tiedot olivat nyt paremmin vertailukelpoisia.



Kuva 42. Kuvakaappaus jään pinnasta, kun Trail Renderer -komponentti oli käytössä.



Kuva 43. Kuvakaappaus jään pinnasta, kun Trail Renderer -komponentti ei ollut käytössä.

Lisäksi testiolosuhteisiin lisäsi hieman satunnaisuutta partikkelisysteemi, joka loi tasoon lumisadetta. Lumisateen määrä pysyi kuitenkin suunnilleen vakiona, minkä vuoksi kerätyt arvotkin vakiintuivat paikoilleen, eikä tätä partikkelisysteemiä tämän vuoksi kannattanut ottaa pois testaamisen ajaksi.

Testitilanteissa pelitilanteiden muutokset minimoitiin siten, että pidin kiekkoa hallussa itse ohjaamallani kenttäpelaajalla, jota en liikuttanut testien aikana ollenkaan[12]. Tällä tavalla varmistettiin se, että testitilanteiden aikana ei syntynyt maaleja, koska syntyneet maalit olisivat tuoneet tarpeettomia muutoksia ruudunpäivitysnopeuteen etenkin, kun aina syntyneen maalin jälkeen peli käynnisti reaktiotesti-minipelin.

12 Tietokoneen ohjaamien kenttäpelaajien tekoäly oli pelissä niin yksinkertainen, että pelaajat osasivat ainoastaan seurata kiekkoa ja laukaista sen pois sen haltuun saadessaan. Kiekon pois ottaminen toisilta pelaajilta ei tietokoneen ohjaamilta kenttäpelaajilta sen sijaan onnistunut.

11.1 Alkutilanne

Puck Buddiesin alkutilanteessa (kuva 44) ei ollut käytetty mitään optimointikeinoja:

- **Valaistus:**
    - Yksi Directional Light -komponentti, jonka dynaamisen valonlähteen tyypiksi oli asetettu automaattinen.
- **Kamera:**
    - Kamera oli asetettu siten, että kaikki pelialueella olevat objektit jouduttiin piirtämään kerralla, minkä lisäksi peliä varten ohjelmoitu dynaaminen kamera oli lukittu paikoilleen.
- **Tekstuurit:**
    - Tekstuurit olivat pakkaamattomia: 3D-malleissa käytetyt kehittyneemmät tekstuurit olivat 32-bittisiä RGBA-tyyppisiä tekstuureita ja muut tekstuurit truecolor-tyyppisiä tekstuureita.
    - Mip-kartoitusta ei ollut käytetty.
- **Fysiikat:**
    - Muunnetun aika-askeleen arvoksi oli asetettu Unityn antama oletusarvo eli 0.02, mikä tarkoitti sitä, että fysiikoita päivitettiin 50 kertaa sekunnissa.
    - Pelaajat käyttivät kapselitörmäyttimiä (engl. capsule collider), kiekko pallotörmäytintä (engl. sphere collider), kaukalon reunat, maalit ja jää 3D-mallin verkon myötäisiä törmäyttimiä ja tukit laatikkotörmäyttimiä. Lisäksi staattiset objektit käyttivät staattisia törmäyttimiä.

Kuva 44. Kuvakaappaus Puck Buddiesista alkutilanteessa, minkä ruudunpäivitysnopeuksia FPS Graph -liitännäinen analysoi kuvan vasemmassa yläkulmassa.

Taulukko 11 kertoo tilastot alkutilanteessa, mistä nähdään, että pelin ruudunpäivitysnopeus oli ilman mitään optimointejakin noin 180 kuvaa sekunnissa.

Taulukko 11. Puck Buddiesin tilastot alkutilanteessa.

ALUSTA	**PC alkutilanteessa**
RESOLUUTIO	1920 x 1080
PIIRTOKUTSUT	94
KOLMIOT	~35 900
TEKSTUURIT	~36.3MB
RUUDUNPÄIVITYS	~180

11.2 Grafiikoiden suorituskyvyn pullonkaula

Aivan kuten Status:Insanenkin grafiikoiden optimoinnin kanssa, myös Puck Buddiesissa testattiin aluksi, parantuiko pelin ruudunpäivitysnopeus resoluutiota laskemalla: taulukko 12 kertoo tilastot, kun pelin resoluutio laskettiin alkutilanteen 1920 x 1080 pikselistä 1280 x 720 pikseliin. Kuten taulukosta nähdään, resoluution laskeminen ei laskenut piirtokutsujen, kolmioiden tai tekstuurien vie-

män muistin määrää ollenkaan. Ruudunpäivitysnopeus nousi kuitenkin 89:llä kuvalla sekunnissa eli 49:llä %:lla verrattuna alkutilanteeseen. Tästä pystyi tekemään sen johtopäätöksen, että grafiikoiden suorituskyvyn pullonkaulana oli todennäköisesti grafiikkaprosessori.

Taulukko 12. Puck Buddiesin tilastot resoluution laskemisen jälkeen.

ALUSTA	**PC nyt**	**PC alkutilanteessa**
RESOLUUTIO	1280 x 720	1920 x 1080
PIIRTOKUTSUT	94	94
KOLMIOT	~35 900	~35 900
TEKSTUURIT	~36.3MB	~36.3MB
RUUDUNPÄIVITYS	~269 (+89)	~180

11.3 Valaistuksen optimointi

Puck Buddies ei sisältänyt lainkaan optimoitavaa dynaamista valaistusta, vaan pelin valaistus tuli kokonaisuudessaan yhdestä Directional Light -komponentista, jolle ei käytetyn piirtämispolun takia voinut tehdä mitään. Tämän vuoksi valaistuksen optimointia ei pystytty tässä pelissä soveltamaan lainkaan.

11.4 Kameran asettelu

Pelin alkutilanteessa kamera oli asetettu siten, että kaikki pelialueella olevat objektit jouduttiin piirtämään kerralla, minkä lisäksi peliä varten ohjelmoitu dynaamisen kamera oli lukittu paikoilleen. Kun kameraa vei lähemmäs pelialuetta, oli sen dynaamiset ominaisuudet otettava takaisin käyttöön, jotta peliä pystyi pelaamaan.

Tässä ongelmana oli kuitenkin se, että kaikissa mitatuissa arvoissa tapahtui jatkuvaa arvojen vaihtelua, mikä oli syynä siihen miksi kameran dynaamiset ominaisuudet oli otettu alunperinkin pois. Mitattuja arvoja ei tällöin pystytty mittaamaan riittävän tarkasti, mutta sen verran pystyttiin kuitenkin kertomaan, että

Status: Insanen tavoin myös tässä pelissä kameran tuomisesta lähemmäksi pelialuetta oli selkeästi hyötyä, koska aina, kun kamera siirtyi lähemmäs pelialuetta, ruudunpäivitysnopeuskin parani hiukan[13].

11.5 Tekstuurien optimointi

Pelin tekstuurit olivat alkutilanteessa pakkaamattomia: 3D-malleissa käytetyt kehittyneemmät tekstuurit olivat 32-bittisiä RGBA-tyyppisiä tekstuureita ja muut tekstuurit truecolor-tyyppisiä tekstuureita. 3D-mallien tekstuureissa ei ollut myöskään käytetty Mip-kartoitusta. Aluksi kokeiltiin, miten paljon tekstuurien pakkaaminen paransi ruudunpäivitysnopeutta: tekstuurit pakattiin käyttämällä kehittyneempiin tekstuureihin automaattista pakkausta, joka valitsi tekstuureille sopivimman vaihtoehdon automaattisesti useiden vaihtoehtojen joukosta ja muihin tekstuureihin tavallista pakkausta, joka oli niille ainoa pakkausvaihtoehto.

Taulukosta 13 nähdään, että tekstuurien pakkaaminen laski tekstuurien viemää muistin määrää noin 80:llä %:lla verrattuna alkutilanteeseen. Ruudunpäivitysnopeus nousi tällöin 35:llä kuvalla sekunnissa eli noin 19:llä %:lla.

Taulukko 13. Puck Buddiesin tilastot, kun käytössä oli tekstuurien pakkaaminen, muttei Mip-kartoitusta.

ALUSTA	PC nyt	PC alkutilanteessa
TEKSTUURIEN PAKKAUS	käytetty	ei käytetty
MIP-KARTOITUS	ei käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080
PIIRTOKUTSUT	94	94
KOLMIOT	~35 900	~35 900
TEKSTUURIT	~7.2MB	~36.3MB
RUUDUNPÄIVITYS	~215 (+35)	~180

13 Ruudunpäivitysnopeuden keskiarvo kameran dynaamisten ominaisuuksien kanssa oli noin 190–195 kuvaa sekunnissa, joka oli noin 10–15 kuvaa enemmän kuin alkutilanteessa.

Seuraavaksi testattiin Mip-kartoitusta käyttämällä Unityn oletusasetuksia. Taulukosta 14 nähdään, että Mip-kartoitus lisäsi tekstuurien viemää muistin määrää noin 16:lla %:lla, mutta ruudunpäivitysnopeus parani kuitenkin 17:lla kuvalla sekunnissa eli noin yhdeksällä prosentilla verrattuna alkutilanteeseen.

Taulukko 14. Puck Buddiesin tilastot, kun tekstuurien pakkaaminen ei ollut käytössä, mutta Mip-kartoitus oli käytössä.

ALUSTA	PC nyt	PC alkutilanteessa
TEKSTUURIEN PAKKAUS	ei käytetty	ei käytetty
MIP-KARTOITUS	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080
PIIRTOKUTSUT	94	94
KOLMIOT	~35 900	~35 900
TEKSTUURIT	~42.2MB	~36.3MB
RUUDUNPÄIVITYS	~197 (+17)	~180

Lopuksi testattiin molempia tekstuurien optimointikeinoja yhdessä. Taulukosta 15 nähdään, että tekstuurien viemä muistin määrä oli tällöin hieman suurempi kuin tilanteessa, jossa käytettiin tekstuurien pakkaamista ilman Mip-kartoitusta. Alkutilanteeseen verrattuna tekstuurien viemä muistin määrä laski kuitenkin 77:llä %:lla. Ruudunpäivitysnopeus oli täysin sama kuin tilanteessa, jossa käytettiin tekstuurien pakkaamista ilman Mip-kartoitusta.

Taulukko 15. Puck Buddiesin tilastot, kun sekä tekstuurien pakkaaminen että Mip-kartoitus olivat käytössä.

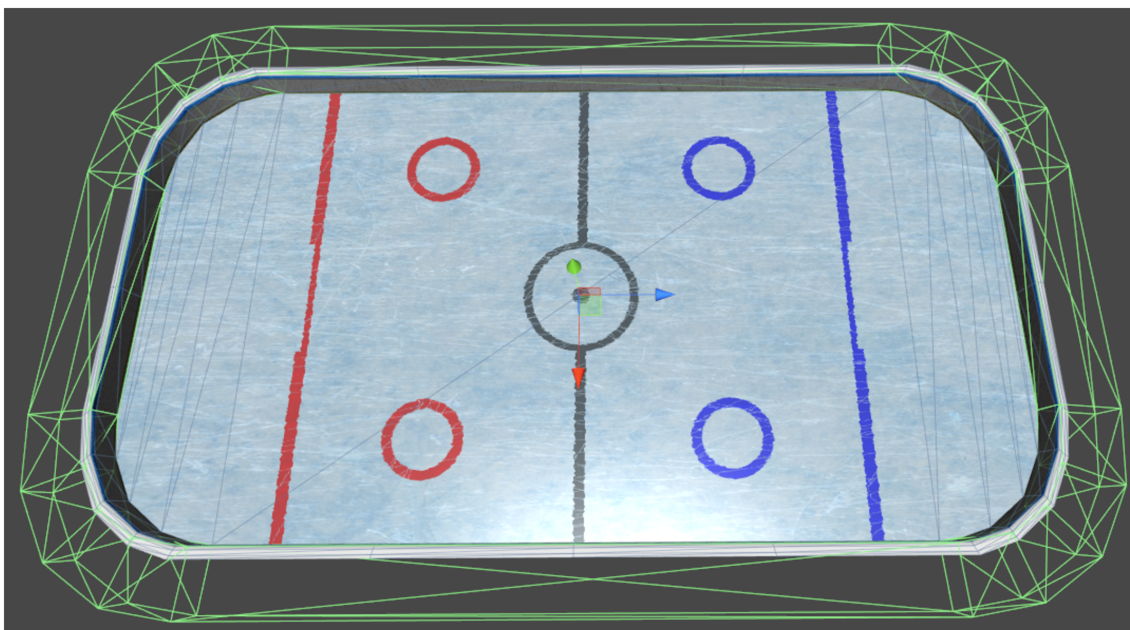
ALUSTA	PC nyt	PC alkutilanteessa
TEKSTUURIEN PAKKAUS	käytetty	ei käytetty
MIP-KARTOITUS	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080
PIIRTOKUTSUT	94	94
KOLMIOT	~35 900	~35 900
TEKSTUURIT	~8.2MB	~36.3MB
RUUDUNPÄIVITYS	~215 (+35)	~180

11.6 Fysiikoiden optimointi

Pelin alkutilanteessa muunnetun aika-askeleen arvoksi oli asetettu Unityn antama oletusarvo eli 0.02, mikä tarkoitti sitä, että fysiikoita päivitettiin 50 kertaa sekunnissa. Aika-askeleen arvo asetettiin arvoon 0.1, jolloin fysiikoita päivitettiin ainoastaan 10 kertaa sekunnissa. Tällä ei kuitenkaan ollut mitään näkyvää vaikutusta ruudunpäivitykseen, minkä lisäksi fysiikoiden alhainen päivitysten määrä sai pelin kenttäpelaajien luistelemisen näyttämään epätasaiselta: pelaajien liikkuminen näytti siltä kuin ne olisivat liikkuneet teleporttaamalla. Tämän lisäksi pelaajat pystyivät Status: Insanen tavoin menemään objektien lävitse, eikä tässäkään tapauksessa ongelmaan auttanut se, että törmäyksien tunnistus asetettiin jatkuvaksi.

Status: Insanessa kaikki käytetyt törmäyttimet olivat laatikkotörmäyttimiä, mutta Puck Buddiesissa erilaisia törmäyttimiä oli käytetty huomattavasti monipuolisemmin: pelaajat käyttivät kapselitörmäyttimiä, kiekko pallotörmäytintä, kaukalon reunat, maalit ja jää 3D-mallin verkon myötäisiä törmäyttimiä ja ainoastaan tukit laatikkotörmäyttimiä. Sen sijaan Status: Insanen tavoin myös tässä pelissä oli käytetty törmäyttimien staattisuutta täysin optimaalisesti hyväksi: dynaamisissa objekteissa, joita olivat kiekko, tukit ja pelaajat, oli käytetty törmäyttimen lisäksi myös jäykkää kappaletta. Sen sijaan staattisissa objekteissa, joita olivat kaukalon reunat, jääkiekkomaalit ja jää, oli käytetty pelkkää törmäytintä ilman jäykkää kappaletta.

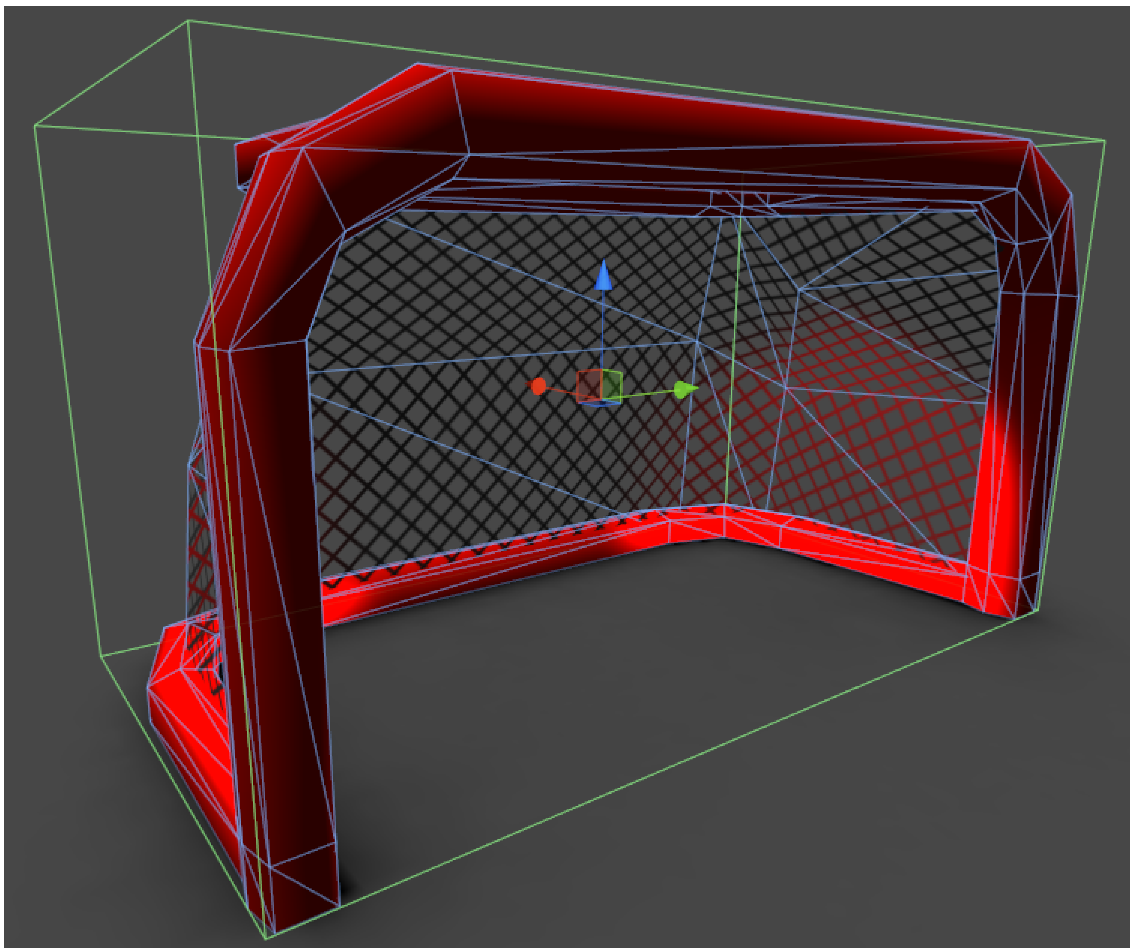
Koska törmäyttimet eivät olleet yksinkertaisimpia mahdollisia, voitiin niitä optimoida. Useimmissa tapauksissa monimutkaisempien törmäyttimien käyttäminen oli tosin perusteltua: esimerkiksi kaukaloiden reunoissa ei olisi ollut järkevää käyttää yksinkertaisempaa törmäytintä (kuva 45).



Kuva 45. Kuvakaappaus Puck Buddiesin kaukalosta, josta nähdään vihreällä törmäyttimen muoto.

Sen sijaan jääkiekkomaaleissa olisi voitu käyttää 3D-mallin verkon myötäisiä törmäyttimiä yksinkertaisempiakin törmäyttimiä, joten ne korvattiin laatikkotörmäyttimillä (kuva 46). Tämä tosin teki maalien tekemisestä mahdotonta, mutta tämä ongelma olisi ollut mahdollista tarpeen tullen ratkaista skriptaamalla[14].

14 Kiekon törmäyskohta olisi ollut mahdollista tunnistaa skriptaamalla (Unity Scripting API 2014i), minkä avulla kiekko olisi ollut mahdollista päästää törmäyttimen läpi maalin etupuolelta.



Kuva 46. Kuvakaappaus Puck Buddiesin jääkiekkomaalista, jossa nähdään sinisillä viivoilla 3D-mallin verkon muoto ja vihreällä törmäyttimen muoto, joka on huomattavasti 3D-mallin verkkoa yksinkertaisempi.

Pelkkä jääkiekkomaalien törmäyttimien vaihtaminen yksinkertaisempiin törmäyttimiin ei kuitenkaan riittänyt nostamaan ruudunpäivitysnopeutta riittävästi, jotta se olisi näkynyt mittaustuloksissa.

12 Tulosten arviointi

Tässä luvussa arvioidaan luvuissa 10 ja 11 saatuja tuloksia. Tulokset arvioidaan testitilannekohtaisesti, jolloin vertaillaan ensiksi Status: Insanen PC- ja Android-versioiden tuloksia, minkä jälkeen vertaillaan Status: Insanen PC-version ja Puck Buddiesin tuloksia. Lopuksi kustakin testitilanteesta on koostettu yhteenveto.

Tuloksia arvioidessa vastattiin seuraaviin kysymyksiin:

- Kuinka tehokkaita optimointikeinot olivat peli- ja alustakohtaisesti?
- Vaikuttivatko optimointikeinot jotenkin pelien grafiikoiden tai muiden ominaisuuksien laatuun?
- Kuinka helppoa optimointikeinoja oli toteuttaa?

12.1 Alkutilanne

Alkutilanne oli tilanne, jossa mitään optimointikeinoja ei ollut käytetty.

12.1.1 Status: Insanen PC- ja Android-versiot

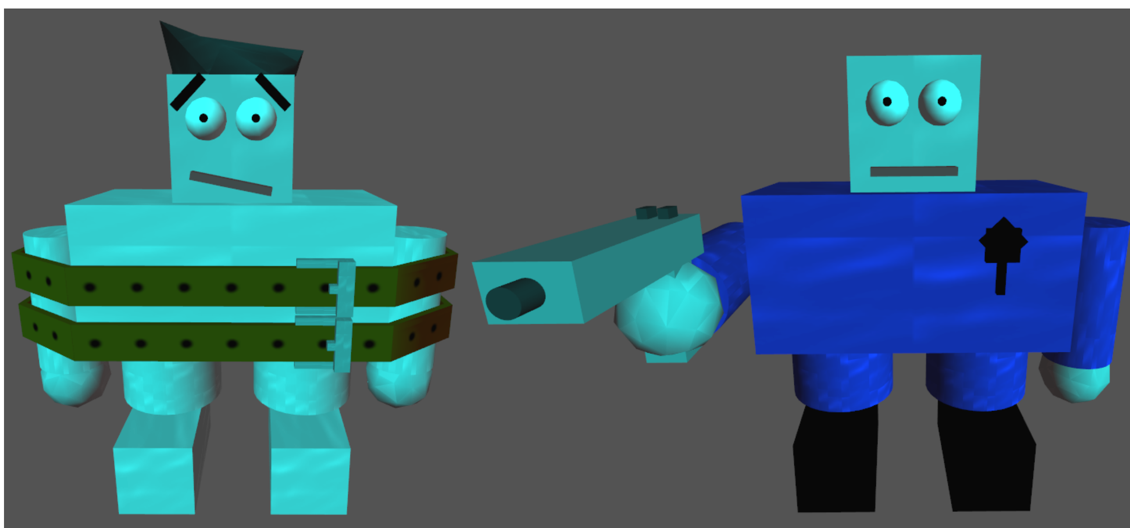
Alkutilanteessa Status: Insanen PC- ja Android-versioiden eroavaisuudet piirtokutsujen määrässä, kolmioiden määrässä ja tekstuurien viemässä muistin määrässä selittyvät täysin käytetyillä laatuasetuksilla: mikäli molemmissa versioissa olisi käytetty samoja laatuasetuksia, nämä arvot olisivat olleet identtisiä. Sen sijaan erot ruudunpäivityksessä selittyvät suurimmaksi osaksi testilaitteiden resurssien eroilla: PC-tietokoneessa oli laitteiston resursseja moninkertainen määrä Android-täppäriin verrattuna.

Versioiden välinen ero ruudunpäivitysnopeuksissa olisikin ollut vielä suurempi, mikäli Androidilla olisi käytetty samoja laatuasetuksia ja samaa resoluutiota kuin PC:llä. Tämä johtuu siitä, että huonommista laatuasetuksista ja alhaisemmasta resoluutiosta oli Androidin ruudunpäivitysnopeudelle ainoastaan hyötyä, koska kuten luvussa 9.4 kerrottiin, Androidilla ruudunpäivitysnopeus olisi ollut PC:n laatuasetuksilla ainoastaan 2 kuvaa sekunnissa ja, kuten luvussa 10.2 kerrottiin, PC-version ruudunpäivitysnopeus parani huomattavasti pelin resoluutiota laskettaessa.

12.1.2 Status: Insanen PC-versio ja Puck Buddies

Status: Insanen PC-versiossa oli alkutilanteessa yli 13-kertainen määrä piirtokutsuja verrattuna Puck Buddiesiin ja piirrettävien kolmioiden määräkin oli melkein kolminkertainen. Sen sijaan tekstuurien viemässä muistin määrässä tilanne oli päinvastainen: Puck Buddiesin tekstuurit veivät yli 13-kertaisen määrän muistia verrattuna Status: Insanen tekstuureihin.

Nämä eroavaisuudet selittyvät ensinnäkin sillä, että Status: Insanessa oli runsaasti pieniä, mutta melko yksinkertaisia ja palikkamaisia objekteja, joissa käytettiin melko suttuisia tekstuureita (kuva 47). Pelin objektimäärää nosti etenkin tason seinät, jotka luotiin Unityn editorissa, koska niiden tekeminen 3D-mallinnuohjelmassa oli liian hankalaa. Tällöin seinät koostettiin pienistä kuutioista, mikä yli kaksinkertaisti pelin objektimäärän. Tämä tarkoitti myös sitä, että piirtokutsujenkin määrä yli kaksinkertaistui. Puck: Buddiesissa sen sijaan oli päinvastaisesti vähän objekteja, mutta ne olivat huomattavasti yksityiskohtaisempia muotonsa ja tekstuurien tarkkuutensa puolesta (kuva 48). Puck Buddiesissa ruudunpäivitysnopeus oli yli kaksi kertaa Status: Insanea suurempi: Puck Buddiesin ruudunpäivitysnopeus oli 180 kuvaa sekunnissa, kun taas Status: Insanessa se oli 75 kuvaa sekunnissa.



Kuva 47. Kuvassa on kaksi Status: Insane -pelin pelihahmoa: vasemmalla puolella on pelaajan ohjaama pelihahmo, joka on pelin monimutkaisin yksittäinen 3D-malli ja oikealla vartija.

Kuva 48. Kuvassa on kaksi Puck Buddies -pelin pelihahmoa: vasemmalla puolella on kenttäpelaaja ja oikealla maalivahti.

Piirtokutsujen, piirrettävien kolmioiden ja ruudunpäivitysnopeuden määrien eroavaisuuksiin vaikutti merkittävästi myös se, että Status: Insanessa oli alkutilanteessa käytössä kaksi pikselikohtaista valaistusta tuottavaa Point Light -komponenttia, kun taas Puck Buddiesissa niitä ei ollut käytössä ainuttakaan. Kun Status: Insanen Point Light -komponenttien dynaamisen valaistuksen tyyppi vaihdettiin pikselikohtaisesta valaistuksesta verteksivalaistukseksi, pelin ruudunpäivitysnopeus nousi 164:ään kuvaan sekunnissa. Puck Buddiesin ruudunpäivitysnopeus oli tällöin enää noin yhdeksän prosenttia suurempi kuin Status: Insanen ruudunpäivitysnopeus. Tällöin myös ero piirrettävien kolmioiden kesken tasoittui: Puck Buddiesissa niitä oli itse asiassa noin neljä prosenttia enemmän. Sen sijaan piirtokutsujen määrä oli Status: Insanessa vieläkin noin 4.6-kertainen verrattuna Puck Buddiesiin johtuen suuremmasta objektien määrästä.

12.2 Grafiikoiden suorituskyvyn pullonkaula

Ennen grafiikoiden optimoinnin aloittamista selvitettiin, oliko grafiikoiden suorituskyvyn pullonkaulana prosessori vai grafiikkaprosessori. Pullonkaula paikannettiin testaamalla alhaisemman resoluution vaikutusta testattavien pelien ruudunpäivitysnopeuksiin. Tätä ei tosin pystytty testaamaan Status: Insanen Android-versiossa ollenkaan, mutta Status: Insanen PC-version ja Puck Buddiesissa kanssa tätä ongelmaa ei ollut.

Molemmissa peleissä saatiin sama tulos: resoluution laskeminen paransi huomattavasti ruudunpäivitysnopeutta, joten suorituskyvyn pullonkaulana oli tällöin molemmissa tapauksissa todennäköisesti grafiikkaprosessori. Tästä tiedosta oli kuitenkin tässä opinnäytetyössä hyvin vähän hyötyä, koska jokaista optimointikeinoa testattiin joka tapauksessa. Toisaalta, useista optimointikeinoista oli hyötyä sekä prosessorille että grafiikkaprosessorille, kuten esimerkiksi piirrettävien objektien määrää vähentäneet optimointikeinot.

Grafiikoiden suorituskyvyn pullonkaula on kuitenkin yksi mahdollisista osatekijöistä, miksi verteksivalaistuksen ja valokartoituksen yhteiskäyttö sai aikaan Status: Insanessa alustakohtaisesti erilaisen lopputuloksen (ks. luku 12.3.1).

12.3 Valaistuksen optimointi

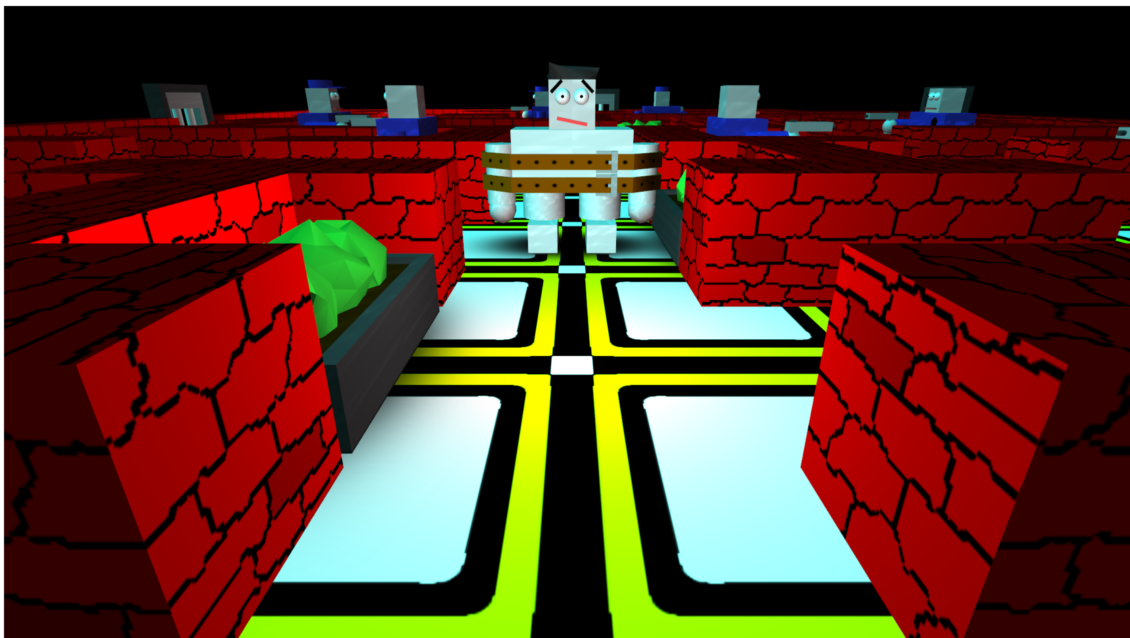
Valaistusta optimoitiin kahdella tavalla: dynaamisen valaistuksen tyyppi pystyttiin vaihtamaan pikselikohtaisesta valaistuksesta verteksivalaistukseksi, minkä lisäksi valaistusta pystyttiin leipomaan.

12.3.1 Status: Insanen PC- ja Android-versiot

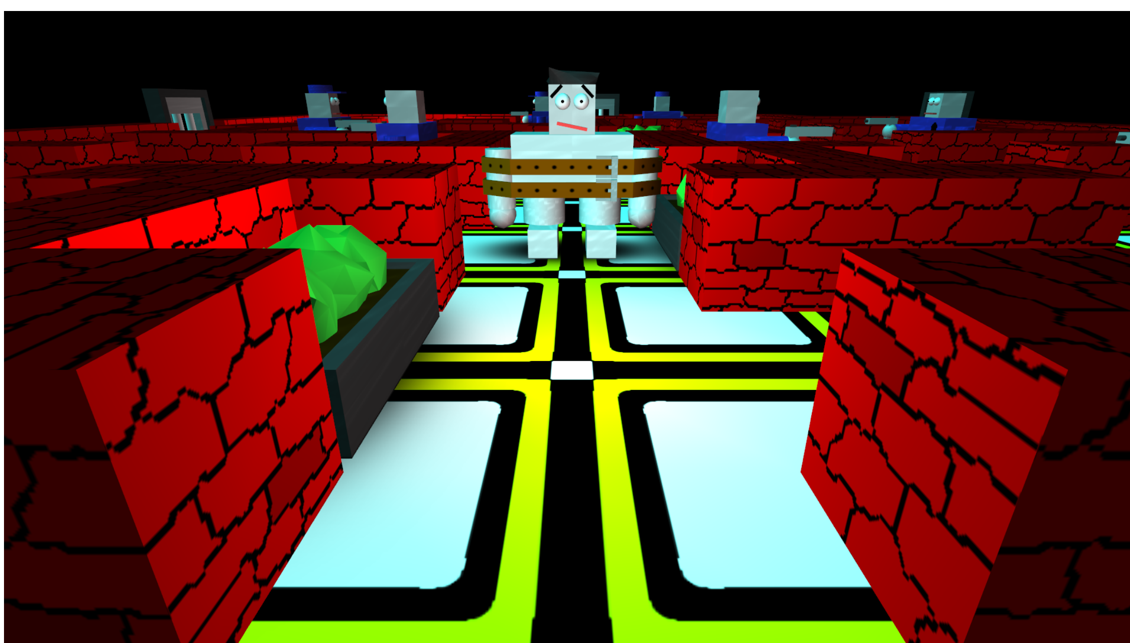
Verteksivalaistus

PC:llä pikselikohtaisten valaistuksen muuttaminen verteksivalaistukseksi laski piirtokutsujen ja piirrettävien kolmioiden määrää paljon sekä nosti ruudunpäivitysnopeuden yli kaksinkertaiseksi alkutilanteeseen nähden. Ruudunpäivitysnopeuden kasvua voidaan pitää yllättävänkin suurena, koska pelissä oli ainoastaan kaksi Point Light -komponenttia, jotka tosin vaikuttivat lähes kaikkiin pelin objekteihin. Tällöin pikselikohtaista valaistusta käytettäessä kaikki objektit jouduttiin piirtämään lähes kolmeen kertaan, kun taas verteksivalaistuksen kanssa ainoastaan kerran.

Kuten luvussa 4.1.1 kerrottiin, verteksivalaistus ei välttämättä ole yhtä näyttävän näköinen kuin pikselikohtainen valaistus, mutta tässä pelissä käytetyllä dynaamisen valaistuksen tyypillä ei ollut juuri mitään vaikutusta pelin grafiikoihin: ainoastaan joissain kaltevissa kulmissa oli havaittavissa pienoisia eroja, kuten kuvia 49 ja kuvia 50 vertailemalla voidaan huomata pelaajan ohjaaman pelihahmon vöistä. Kannattaa kuitenkin muistaa, että luvussa 4.1.1 nähdyssä esimerkissä Spot Light -komponenttien kanssa (joita siis Status: Insanessa ei ollut käytössä ainuttakaan) dynaamisen valaistuksen tyypillä oli suuri merkitys, minkä lisäksi myös jotkin tehosteet ovat mahdollisia ainoastaan pikselkohtaisesti piirrettynä. Täten pikselikohtaista valaistusta ei voida pitää resurssien tuhlaamisena, mikäli sille on oikeasti tarvetta.



Kuva 49. Kuvakaappaus Status: Insanen PC-version viidennestä tasosta, kun käytössä on pikselikohtainen valaistus. Kuvakaappauksessa toinen tason kahdesta Point Light -komponenteista on aivan pelaajan ohjaaman pelihahmon edessä.



Kuva 50. Kuvakaappaus Status: Insanen PC-version viidennestä tasosta, kun käytössä on verteksivalaistus. Kuvakaappauksessa toinen tason kahdesta Point Light -komponenteista on aivan pelaajan ohjaaman pelihahmon edessä.

Valaistuksen vaihtaminen pikselikohtaisesta valaistuksesta verteksivalaistukseksi oli todella helppoa, koska ainut asia mitä tarvitsi tehdä, oli säätää laatuasetuksista pikselikohtaisten valonlähteiden maksimimäärä nollaan. Androidilla

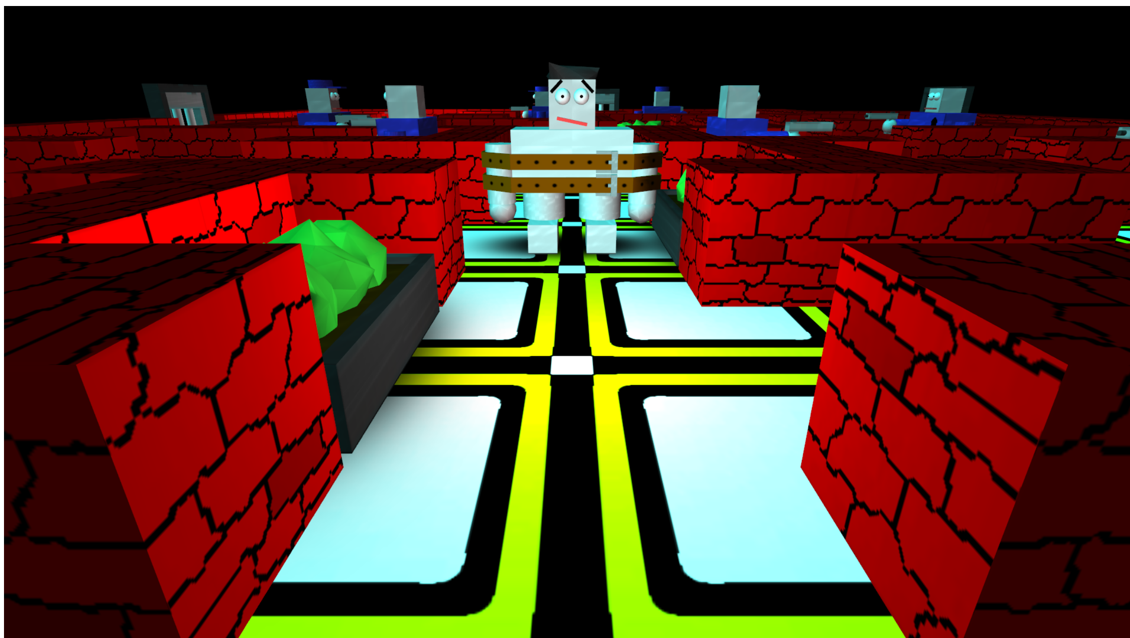
pikselikohtaisten valonlähteiden maksimimäärä oli asetettu nollaan laatuasetuksista jo alkutilanteessa, mikä selittää täysin sen, miksi Androidilla ei tässä vaiheessa tapahtunut mitään muutoksia.

Valokartoitus

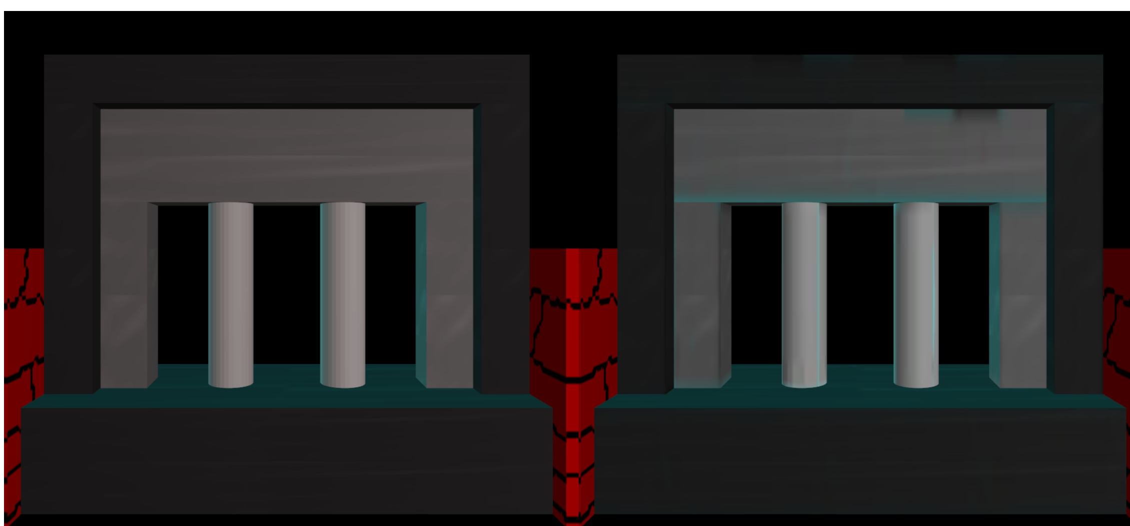
Valokartoitus vaikutti PC:llä tilastoihin kaksijakoisesti: toisaalta piirtokutsujen määrä ja piirrettävien kolmioiden määrä tippuivat, mutta toisaalta tekstuurien viemä muistin määrä nousi merkittävästi. Valokartoituksesta oli kuitenkin enemmän hyötyä kuin haittaa, koska ruudunpäivitysnopeus nousi noin 15:llä %:lla verrattuna alkutilanteeseen.

Androidilla sen sijaan ainoat tilastoissa tapahtuneet muutokset olivat vähäinen tekstuurien viemän muistin määrän nouseminen ruudunpäivitysnopeuden paraneminen noin 24:llä %:lla. Se, että tilastojen huonontumisesta huolimatta ruudunpäivitys kuitenkin parantui, johtuu siitä, että vähäisempi verteksivalaistuksen käyttäminen ei näy mitenkään tilastoissa: verteksivalaistus ei vaikuttanut piirtokutsujen, kolmioiden tai tekstuurien viemään muistin määrään mitenkään, vaan se vähensi ainoastaan jokaisella ruudunpäivityksellä tehtäviä valaistuksen laskemiseen tarvittavia laskutoimituksia.

Valokartoituksen vaikutusta pelin grafiikoihin ei juurikaan huomaa useimmista staattisista objekteista, kuten seinistä ja laittiasta (kuva 51), mutta etenkin ikkunoissa näkyi pieniä eroja mikäli niitä tarkasteltiin todella läheltä (kuva 52). Se, että huononsiko valokartoitus varsinaisesti valaistuksen laatua, lienee makuasia, mutta ainakaan sitä ei voi kaikissa tilanteissa pitää täysin neutraalina optimointikeinona.



Kuva 51. Kuvakaappaus Status: Insanen PC-version viidennestä tasosta, kun staattisten objektien valaistus on leivottu käyttäen valokartoitusta. Kuvakaappauksessa toinen tason kahdesta Point Light -komponenteista on aivan pelaajan ohjaaman pelihahmon edessä.



Kuva 52. Kuvakaappaus Status: Insanen ikkunoista, joista vasemmanpuoleinen käyttää pikselikohtaista valaistusta ja oikeanpuoleisen valaistus on leivottu käyttäen valokartoitusta.

Valokartoituksen alkuvalmistelut oli helppo tehdä, minkä jälkeen valaistuksen leipominen ei vaatinut enää kuin muutaman hiiren klikkauksen, joten myös valokartoituksen tekemistä voi luonnehtia helpoksi optimointikeinoksi.

Verteksivalaistuksen ja valokartoituksen käyttäminen yhdessä

PC:llä verteksivalaistuksen ja valokartoituksen käyttäminen yhdessä paransi ruudunpäivitysnopeutta yli kaksinkertaiseksi verrattuna alkutilanteeseen. Ruudunpäivitysnopeus oli tosin vielä hieman parempi, kun verteksivalaistusta käytettiin ilman valokartoitusta, joten valokartoituksen aiheuttamasta tekstuurien viemän muistin määrän kasvusta aiheutuu tässä tapauksessa enemmän haittaa kuin verteksivalaistuksen vähenemisestä on hyötyä. Kuten tuli todettua aiemmin, Androidilla vaikutus oli päinvastainen, minkä näkee taulukosta 16.

Taulukko 16. Status: Insanen tilastot, joissa vertaillaan kahta optimointitilannetta, joista toisessa on käytetty verteksivalaistusta ilman valokartoitusta ja toisessa valokartoituksen kanssa. Taulukosta nähdään, että verteksivalaistuksen yhteisvaikutus valokartoituksen kanssa vaikutti ruudunpäivitysnopeuteen PC:llä päinvastaisesti Androidiin verrattuna.

ALUSTA	**PC**	**PC**	**Android**	**Android**
DYNAAMISEN VALAISTUKSEN TYYPPI	verteksivalaistus	verteksivalaistus	verteksivalaistus	verteksivalaistus
VALOKARTOITUS	käytetty	ei käytetty	käytetty	ei käytetty
RESOLUUTIO	1920 x 1080	1920 x 1080	1024 x 600	1024 x 600
PIIRTOKUTSUT	425	425	425	425
KOLMIOT	~34 500	~34 500	~34 500	~34 500
TEKSTUURIT	~10.7MB	~2.7MB	~204.7KB	~140.8KB
RUUDUNPÄIVITYS	~155 (-9)	~164	~21 (+4)	~17

Paras keksimäni selitys, miksi näin tapahtui on se, että PC:llä tekstuurien viemä muistin määrä kasvoi noin kahdeksalla megatavulla eli 296:lla %:lla, kun taas Androidilla kasvua oli ainoastaan noin 63.9 kilotavua eli 45 %:a. Koska tekstuurien viemän muistin määrän kasvun ero eri alustojen kesken oli noin merkittävä, on täysin mahdollista, että se aiheutti kyseisen omituisuuden tilastoissa.

Toinen selitys liittyy grafiikoiden suorituskyvyn pullonkaulaan: PC:llähän se oli todennäköisesti grafiikkaprosessori, kun taas Androidilla sitä ei pystytty mittaamaan ollenkaan. Tällöin on mahdollista, että testilaitteiden kesken pullonkauloissa on ero. Tätä selitystä tukee se, että piirrettävien tekstuurien viemän muistin määrän kasvu kuormittaa juurikin grafiikkaprosessoria.

Joka tapauksessa valokartoituksesta voi olla verteksivalaistuksen kanssa yhdessä käytettynä jopa hieman haittaa ruudunpäivitysnopeudelle. Tällöin valokartoitusta ei kannattaisi käyttää, jollei sillä saavuteta jotain subjektiivista grafiikoiden laadun paranemista.

12.3.2 Status: Insanen PC-versio ja Puck Buddies

Puck Buddiesissa ei ollut käytetty lainkaan dynaamista valaistusta, minkä vuoksi sen valaistusta ei pystytty optimoimaan, joten mitään vertailujakaan Status: Insanen kanssa ei voida tehdä.

12.3.3 Yhteenveto

Valaistus näyttäisi vaikuttavan pelien suorituskykyyn todella merkittävästi, minkä lisäksi valaistuksen optimointi on tehty Unityssä todella helpoksi. Tämän vuoksi valaistus näyttäisi olevan yksi parhaista optimointikohteista, kunhan muistaa, että valaistuksen optimointi voi vaikuttaa hieman grafiikoiden laatuun ja, että valokartoitus voi verteksivalaistuksen kanssa yhdessä käytettynä hieman huonontaa ruudunpäivitysnopeutta.

12.4 Kameran asettelu

Kameran asettelussa testattiin, vaikuttaako kameran etäisyys pelien suorituskykyyn.

12.4.1 Status: Insanen PC- ja Android-versiot

Kameran vieminen lähemmäksi pelialuetta paransi ruudunpäivitysnopeutta PC:llä noin 15:llä %:lla ja Androidilla noin 35:llä %:lla verrattuna alkutilanteeseen, mikä oli hyvä tulos etenkin Androidilla, koska se oli täten alustan tehokkain yksittäinen testattu optimointikeino. Tuloksissa ei ollut mitään yllättävää: kun objekteja tarvitsi piirtää vähemmän, tarvitsi prosessorin lähettää piirtokutsuja grafiikkarajapinnalle vähemmän. Koska piirrettäviä objekteja oli vähemmän, myös piirrettäviä kolmioita oli luonnollisesti vähemmän. Tämän lisäksi tekstuureiden viemä muistin määrä laski, koska tekstuureja tarvittiin vähemmän. Tästä seurasi vääjäämättömästi se, että ruudunpäivitysnopeus nousi.

Status: Insanessa kameran vieminen lähemmäksi pelialuetta oli varsin helppo toteuttaa, koska kameran siirtäminen ei ollut vaikeaa, mutta sillä oli tässä pelissä kaksi haittavaikutusta:

- Kamera ei enää piirtänyt koko pelialuetta kerralla, joten siitä piti ohjelmoida pelaajan ohjaamaa pelihahmoa seuraava, koska muutoin pelin pelaaminen olisi ollut mahdotonta.
- Kameraan piti lisätä ominaisuus, jolla pelaaja pystyi halutessaan liikuttamaan kameraa myös itse ilman, että pelaajan ohjaama pelihahmo liikkui. Ilman tuota ominaisuutta peli olisi ollut epäreilu, koska vartijat pystyivät näkemään pelaajan ohjaaman pelihahmon vaikka ne olisivatkin kameran ulkopuolella, kun taas pelaajalla ei tällöin olisi ollut mitään mahdollisuutta nähdä niitä.

12.4.2 Status: Insanen PC-versio ja Puck Buddies

Puck Buddiesin kanssa kameran asettelua ei voitu mitata tarkasti, mutta kameran tullessa lähemmäksi pelialuetta ruudunpäivitysnopeus parani myös siinä selkeästi hiukan. Tuloksen epätarkkuuden vuoksi tarkka tulosten tarkka vertailu Status: Insanen kanssa ei kuitenkaan onnistu.

12.4.3 Yhteenveto

Kameran vieminen lähemmäksi pelialuetta näyttäisi parantavan ruudunpäivitysnopeutta pelistä ja alustasta riippumatta, mutta samalla voidaan saada aikaiseksi hankalia ongelmia, joten kameran asettelua ei voida luonnehtia erityisen helpoksi optimointikeinoksi ainakaan kaikissa tilanteissa.

12.5 Tekstuurien optimointi

Tekstuureja optimoitiin kahdella tavalla: tekstuureita pystyttiin pakkaamaan, minkä lisäksi niille pystyttiin tekemään Mip-kartoitus.

12.5.1 Status: Insanen PC- ja Android-versiot

Tekstuurien pakkaaminen

Tekstuurien pakkaaminen laski tekstuurien viemää muistin määrää merkittävästi molemmilla alustoilla, minkä ansiosta grafiikkaprosessori todennäköisesti kuormittui vähemmän. Tämän seurauksena ruudunpäivitysnopeuskin parani hiukan.

Mip-kartoitus

PC:llä kaikki tilastot ruudunpäivitysnopeutta myöten olivat täysin samat kuin alkutilanteessa. Todennäköisesti se ei ollut koko totuus, vaan tilastojen muuttumattomuus johtui myös siitä, että Rendering Statistics Window pyöristi hieman tekstuurien viemän muistin määrän arvoja. Tätä olettamusta tukee se, että Androidilla tekstuurien käyttämässä muistin määrässä oli nähtävissä pieni ero alkutilanteeseen verrattuna. Kummallakaan alustalla ruudunpäivitysnopeus ei kuitenkaan muuttunut, mikä johtunee ainakin osittain siitä, että Mip-kartoitettujen tekstuurien koot olivat hyvin pieniä.

Tekstuurien pakkaamisen ja Mip-kartoituksen käyttäminen yhdessä

PC:llä tilastot olivat täysin samat kuin ne olivat, kun käytettiin tekstuurien pakkaamista, muttei Mip-kartoitusta, mikä todennäköisesti ei tässäkään tapauksessa ollut koko totuus, vaan tekstuurien viemän muistin määrän muuttumattomuus johtunee osittain arvojen pyöristämisestä. Tätä olettamusta tukee tälläkin kertaa se, että Androidilla tekstuurien viemä muistin määrä oli hieman alhaisempi kuin se oli, kun käytettiin tekstuurien pakkaamista, muttei Mip-kartoitusta. Siihen tilanteeseen verrattuna ruudunpäivitysnopeudessa ei ollut muutosta myöskään Androidilla, joten Mip-kartotuksesta ei ollut Status: Insanessa mitään mitattavaa hyötyä tai haittaa kummallakaan alustalla.

12.5.2 Status: Insanen PC-versio ja Puck Buddies

Tekstuurien pakkaaminen

Puck Buddiesissa tekstuurien pakkaaminen paransi ruudunpäivitysnopeutta 19:llä %:lla, kun taas Status: Insanessa hyöty jäi ainoastaan viiteen prosenttiin, mikä johtunee Stutus: Insanen pienemmistä tekstuureista.

Mip-kartoitus

Puck Buddiesissa Mip-kartoitus lisäsi tekstuurien viemää muistin määrää noin 16:lla %:lla, mutta ruudunpäivitysnopeus parani kuitenkin noin yhdeksällä prosentilla alkutilanteeseen verrattuna, kun taas Status: Insanessa ei tapahtunut mitään muutoksia. Tämäkin johtuu todennäköisesti tekstuurien kokoeroista.

Tekstuurien pakkaamisen ja Mip-kartoituksen käyttäminen yhdessä

Puck Buddiesilla tekstuurien viemä muistin määrä oli hieman suurempi kuin tilanteessa, jossa käytettiin tekstuurien pakkaamista ilman Mip-kartoitusta. Ruudunpäivitysnopeus oli täysin sama kuin tilanteessa, jossa käytettiin tekstuurien pakkaamista ilman Mip-kartoitusta. Tästä voi päätellä, että Mip-kartoituksen hyöty vähenee samalla, kun tekstuureiden koko pienenee.

12.5.3 Yhteenveto

Tekstuurien pakkaaminen paransi ruudunpäivitysnopeutta edes vähän kaikissa tapauksissa, joten sitä voi pitää melko perusvarmana optimointikeinona, jolla ei todennäköisesti voi huonontaa pelien ruudunpäivitysnopeutta. Myöskään grafiikoiden laatuun sillä ei ollut silmin nähtävää muutosta, mikä johtui mahdollisesti osittain siitä, että Unity teki pakkaamiset automaattisesti, jolloin se valitsi usean pakkausvaihtoehdon joukosta sopivimman.

Mip-kartoituksen hyöty oli joko täysin olematon tai vähäinen, mutta ainakaan siitä ei testattavissa peleissä ollut haittaa. Sekin voi tosin olla mahdollista, kuten luvussa 4.2.1 kerrottiin. Molempien tekstuurien optimointikeinojen hyödyllisyyteen näyttäisi vaikuttavan tekstuurien koko: suurille tekstuureille hyötyä on enemmän. Molemmat tekstuurien optimointikeinoista olivat hyvin helppoja toteuttaa, koska niiden käyttäminen tarvitsi ainoastaan sallia tekstuurien tuontiasetuksista.

12.6 Fysiikoiden optimointi

Fysiikoita optimoitiin kahdella tavalla: ensiksi testattiin muunnetun aika-askeleen vaikutusta pelien suorituskykyyn, minkä lisäksi testattiin törmäyttimien optimointia.

Muunnettu aika-askel

Muunnetun aika-askeleen arvon nostamisella ei saatu kummassakaan pelissä parannusta ruudunpäivitysnopeuteen, vaan ainoa ero oli se, että sillä aiheutettiin peleihin bugeja erityisesti törmäyksien tunnistamisen kanssa. Edes törmäyksien tunnistamisen asettaminen jatkuvaksi ei tähän ongelmaan auttanut kummassakaan pelissä. Törmäyksien tunnistamisen bugeihin löytyy toki ratkaisuksi esimerkiksi Brauerin ja Adrianin tekemä DontGoThroughThings-skripti (2012), jonka toimivuutta ei keritty testaamaan.

Joka tapauksessa tällä optimointikeinolla ei ollut testattavien pelien ruudunpäivitysnopeuksiin mitään vaikutusta, mikä johtunee ainakin osittain siitä, etteivät pelit olleet kovinkaan fysiikkapainotteisia. Fysiikkapainotteisemmissa peleissä tätä optimointikeinoa kannattanee siis edelleen harkita. Muunnetun aika-askeleen säätäminen oli todella helppoa, mutta mikäli törmäyksien tunnistuksen bugeja jouduttaisiin korjaamaan skriptaamalla, ei tätä optimointikeinoa voitaisi silloin pitää kovin helppona optimointikeinona.

Törmäyttimet

Törmäyttimien optimoimista ei pystytty Status: Insanessa testaamaan ollenkaan, kun taas Puck Buddiesissa siitä ei ollut riittävästi hyötyä, jotta sillä olisi ollut mitään vaikutusta ruudunpäivitysnopeuteen. Puck Buddiesissa oli tosin käytetty törmäyttimiä pääasiallisesti hyvin järkevästi, eikä pelissä ollut ainuttakaan todella monimutkaista törmäytintä. Täten on täysin mahdollista, että todella monimutkaisilla 3D-mallin verkon myötäisillä törmäyttimillä olisi ollut ruudunpäivitysnopeutta huomattavasti alentava vaikutus, minkä vuoksi niitä olisi kannattanut optimoida.

Törmäyttimien optimointi on varsin helppoa, koska Unity osaa asettaa törmäyttimet oletuksellisesti siten, että ne peittävät 3D-mallit alleen mahdollisimman sopivasti. Tämän jälkeen törmäyttimien tarkempikaan asettelu ei ole vaikeaa. Unitystä ei kuitenkaan löydy kuin rajoitetusti erilaisia törmäyttimiä, joten mikäli sopivaa ei niiden joukosta löydy, sellainen joudutaan tekemään 3D-mallinnusohjelmalla, jolloin optimoinnin vaikeusaste todennäköisesti kohoaisi.

13 Loppusanat

Optimointitesteistä kävi ilmi, että etenkin valaistus voi vaikuttaa pelien suorituskykyyn todella paljon. Lähes kaikilla muillakin optimointikeinoilla saatiin pelien suorituskykyyn jonkin verran parannuksia aikaan. Säännön ainoa poikkeus oli fysiikat, joita optimoimalla ei saatu aikaiseksi ollenkaan tuloksia, jotka olisivat ol-

leet riittävän merkittäviä, jotta ne olisivat näkyneet mittaustuloksissa. Testattavia pelejä oli tosin ainoastaan kaksi kappaletta, minkä vuoksi etenkin juuri fysiikoiden optimoinnissa saatuihin tuloksiin tulee suhtautua varauksella, koska kumpikaan peleistä ei ollut erityisen fysiikkapainotteinen: on täysin mahdollista, että jossain testipelejä huomattavasti fysiikkapohjaisemmassa pelissä fysiikoilla olisi paljon suurempi vaikutus ruudunpäivitysnopeuteen.

Testatut optimointikeinot olivat suurimmaksi osaksi helppoja toteuttaa, koska Unityssä on runsaasti erilaisia sisäänrakennettuja ominaisuuksia. Tämän ansiosta esimerkiksi tekstuurien pakkaminen, Mip-kartoitus ja valokartoitus saatiin kätevästi tehtyä Unityssä ilman, että olisi tarvinnut etsiä itse erillisiä liitännäisiä tai muita tietokoneohjelmia Internetistä. Ainoastaan ruudunpäivitysnopeuksien keskiarvojen mittaamiseksi tarvitsi etsiä erillinen liitännäinen itse. Lisäksi kameran asettelu oli muita optimointikeinoja hieman haastavampi toteuttaa, mutta myös fysiikoiden optimoimisen aiheuttamat mahdolliset bugit törmäyksien tunnistamisissa olisivat todennäköisesti vaatineet hieman aikaa ja vaivaa (tämän opinnäytetyön puitteissa niitä ei koetettu ratkaista).

Kaikkia teoriaosiossa mainituista optimointikeinoista ei kuitenkaan pystytty testaamaan käytännön osiossa. Näistä 3D-mallien muodon yksinkertaistaminen ja objektimäärän pitäminen sopivana ovat todennäköisesti varsin hyviä periaatteita 3D-malleja mallinnettaessa. Tarpeettomien ruudunpäivityksen välttämistä sen sijaan voidaan pitää hyvänä periaatteena koodeja ohjelmoitaessa. Objektien altausta kannattanee harkita, mikäli pelissä on runsaasti objekteja, joita luodaan paljon ja joiden elinikä on lyhyt.

Kuten luvussa 3 kerrottiin, pelien optimointien suunnittelussa tulee ottaa huomioon kohdealusta ja, mikäli kohdealustasta löytyy paljon tehoja verrattuna pelin vaatimuksiin, ei optimointi ole kovinkaan tärkeää. Saatujen tulosten perusteella PC:llä optimointi ei ole välttämättä kovinkaan tärkeää, koska sekä Status: Insanen että Puck Buddiesin ruudunpäivitysnopeus oli jo ilman optimointejakin yli 60 kuvaa sekunnissa[15]. Molemmat testipeleistä olivat tosin melko yksinkertai-

15 60 kuvaa sekunnissa -ruudunpäivitysnopeus on PC-peleissä hyvin yleinen standardi, kun

sia indie-pelejä, eivätkä suuren budjetin AAA-pelejä, joissa optimointi olisi ollut todennäköisesti tärkeämpää: esimerkiksi vastikään pelaamani AAA-pelin, Batman: Originsin (Warner Bros. Interactive Entertainment 2013), ruudunpäivitysnopeus nyki testilaitteena käytetyllä PC-tietokoneella jonkin verran, kun koetin pelata sitä parhaalla laadulla.

Toisaalta, testilaitteena ollut PC-tietokone ei ollut aivan uusi, pelaamista varten varta vasten kasattu tehokone, vaan hieman yli kaksi vuotta vanha pakettitietokone. Toisaalta, mikäli pelin kohderyhmä halutaan pitää mahdollisimman laajana, tulisi peli saada toimimaan myös markkinoiden tehokkaimpia laitteita vähemmän tehokkailla laitteilla riittävän hyvin, koska kaikilla pelin potentiaalisilla pelaajilla ei markkinoiden tehokkaimpia laitteita käytössään todennäköisesti ole. Täten optimointi voi olla melko tarpeellista myös PC:llä. Status: Insanen Android-versio sen sijaan osoitti, että mobiililaitteilla optimointi voi olla tärkeää jo tämän mittakaavan peleissä, koska sen ruudunpäivitysnopeus jäi alkutilanteessa kauaksi 60:stä kuvasta sekunnissa.

---

taas esimerkiksi konsolipeleissä käytetään usein alhaisempiakin ruudunpäivitysnopeuksia (starkka 2014).

Lähteet

Activision. 1982. Pitfall!.
Aleksandr. 2014. Documentation, Unity scripting languages and you. Technology – Unity Blog. http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/. 9.11.2014.
Arnold, T. 2012. pitfall-screenshot2.jpg. http://www.retrodomination.com/wp-content/uploads/2012/10/pitfall-screenshot2.jpg.18.11.2014.
Atari Inc.. 1972. Pong.
Brauer, D & Adrian. 2012. DontGoThroughThings. http://wiki.unity3d.com/index.php?title=DontGoThroughThings. 17.11.2014.
Calleja, S. 2006. Space Invader Cabinet.jpg. http://www.flickr.com/photos/scalleja/162952563/in/photostream/. 12.11.2014.
Dented Pixel. 2014a. FPS Graph 0.975.
Dented Pixel. 2014b. FPS Graph - Performance Analyzer. http://u3d.as/content/dented-pixel/fps-graph-performance-analyzer/400. 28.10.2014.
EA Sports. 1991–2014. NHL-pelisarja.
Eidos Interactive. 2000. Deus Ex.
Electronic Arts. 2009. Dragon Age: Origins.
Flight Systems LLC. 2012. Flight Unlimited Las Vegas.
Georgiou, M. 2012. Mip Maps Are Over Rated In Unity3D, Save 33% Texture Memory!. Unity3D Optimization Techniques. http://www.mel-georgiou.co.uk/mip-maps-unity3d-texture-memory/. 4.11.2014.
Honkala, T. 2012a. Legendaariset pelit: Pitfall!. Teoksessa Pelit 1/2012. Helsinki: Sanoma Magazines Finland Oy, 62–65.
Honkala, T. 2012b. Legendaariset pelit: Deus Ex. Teoksessa Pelit 3/2012. Helsinki: Sanoma Magazines Finland Oy, 70–73.
Kuorikoski, J. 2014. Sinivalkoinen pelikirja – Suomen pelialan kronikka 1984–2014. Kustantajan kotipaikka tuntematon: FOBOS.
Lomas, N. 2014. The Console Market Is In Crisis. http://techcrunch.com/2014/03/09/console-crisis/. 28.10.2014.
LucasArts. 2003. Star Wars: Knights of the Old Republic.
Madfinger Games. 2011. Shadowgun.
Microsoft Game Studios. 2007. Mass Effect.
Mossmouth. 2009. Spelunky.
NowGamer. 2011. 9 Ways Deus Ex Changed Gaming Forever. http://www.nowgamer.com/9-ways-deus-ex-changed-gaming-forever. 28.9.2014.
Ready Up. 2010. deusex_bemutato_05.jpg. http://ready-up.net/wp-content/uploads/2010/06/deusex_bemutato_05.jpg. 18.11..2014.
starkka. 2014. 60 FPS on Consoles. http://www.giantbomb.com/60-fps-on-consoles/3015-3223/. 16.11.2014.

Stockton, K. 2012. Sky Castle.
Taito Corporation. 1978. Space Invaders.
Ubisoft. 2002–2013. Tom Clancy's Splinter Cell -pelisarja.
Unity Manual. 2012a. Practical Guide to Optimization for Mobiles - Scripting and Gameplay Methods. http://docs.unity3d.com/412/Documentation/Manual/iphone-OptimizedScriptingMethods.html. 6.3.2014.
Unity Manual. 2012b. Practical Guide to Optimization for Mobiles - Graphics Methods. http://docs.unity3d.com/412/Documentation/Manual/iphone-OptimizedGraphicsMethods.html. 6.3.2014.
Unity Manual. 2012c. Practical Guide to Optimization for Mobiles - Optimizing Scripts. http://docs.unity3d.com/412/Documentation/Manual/iphone-PracticalScriptingOptimizations.html. 6.3.2014.
Unity Manual. 2014a. 2D Textures. http://docs.unity3d.com/Manual/class-TextureImporter.html. 25.10.2014.
Unity Manual. 2014b. Reducing the File Size of the Build. http://docs.unity3d.com/Manual/ReducingFilesize.html. 24.10.2014.
Unity Manual. 2014c. Optimizing Graphics Performance. http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html. 24.10.2014.
Unity Manual. 2014d. Rendering Paths. http://docs.unity3d.com/Manual/RenderingPaths.html. 9.11.2014.
Unity Manual. 2014e. Light. http://docs.unity3d.com/Manual/class-Light.html. 6.11.2014.
Unity Manual. 2014f. Lightmapping Quickstart. http://docs.unity3d.com/Manual/Lightmapping.html. 25.10.2014.
Unity Manual. 2014g. Lightmapping In-Depth. http://docs.unity3d.com/Manual/LightmappingInDepth.html. 16.11.2014.
Unity Manual. 2014h. 3D Textures. http://docs.unity3d.com/Manual/class-Texture3D.html. 25.10.2014.
Unity Manual. 2014i. Time Manager. http://docs.unity3d.com/Manual/class-TimeManager.html. 25.10.2014.
Unity Manual. 2014j. Rigidbody. http://docs.unity3d.com/Manual/class-Rigidbody.html. 9.11.2014.
Unity Manual. 2014k. Colliders. http://docs.unity3d.com/Manual/CollidersOverview.html. 25.10.2014.
Unity Manual. 2014l. Rigidbody 2D. http://docs.unity3d.com/Manual/class-Rigidbody2D.html. 10.11.2014.
Unity Manual. 2014m. Profiler. http://docs.unity3d.com/Manual/Profiler.html. 29.10.2014.
Unity Manual. 2014n. Rendering Statistics Window. http://docs.unity3d.com/Manual/RenderingStatistics.html. 27.10.2014.
Unity Manual. 2014o. Quality Settings. http://docs.unity3d.com/Manual/class-QualitySettings.html. 25.10.2014.
Unity Manual. 2014p. Shadows in Unity. http://docs.unity3d.com/Manual/Shadows.html. 16.11.2014.

Unity Manual. 2014q. Level of Detail. http://docs.unity3d.com/Manual/LevelOfDetail.html. 16.11.2014.
Unity Manual. 2014r. Particle system. http://docs.unity3d.com/Manual/class-ParticleSystem.html.16.11.2014.
Unity Manual. 2014s. Player Settings. http://docs.unity3d.com/Manual/class-PlayerSettings.html. 6.11.2014.
Unity Manual. 2014t. Forward Rendering Path Details. http://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html. 13.11.2014.
Unity Scripting API. 2014a. Rigidbody. http://docs.unity3d.com/ScriptReference/Rigidbody.html. 3.11.2014.
Unity Scripting API. 2014b. MonoBehaviour.OnBecameVisible(). http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnBecameVisible.html. 25.10.2014.
Unity Scripting API. 2014c. MonoBehaviour.OnBecameInvisible(). http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnBecameInvisible.html. 25.10.2014.
Unity Scripting API. 2014d. MonoBehaviour.Update(). https://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.html. 25.10.2014.
Unity Scripting API. 2014e. MonoBehaviour.FixedUpdate(). http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html. 25.10.2014.
Unity Scripting API. 2014f. MonoBehaviour.OnGUI(). https://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnGUI.html. 25.10.2014.
Unity Scripting API. 2014g. Vektor2. http://docs.unity3d.com/ScriptReference/Vector2.html. 9.11.2014.
Unity Scripting API. 2014h. Vektor3. http://docs.unity3d.com/ScriptReference/Vector3.html. 9.11.2014.
Unity Scripting API. 2014i. Collider.OnCollisionEnter(Collision). http://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html. 21.11.2014.
Unity Scripting Reference. Overview: Performance Optimization. http://docs.unity3d.com/410/Documentation/ScriptReference/index.Performance_Optimization.html. 25.10.2014.
Unity Technologies. 2014a. Unity 4.5.4f1.
Unity Technologies. 2014b. Effortlessly unleash your game on the world's hottest platforms. https://unity3d.com/unity/multiplatform. 8.11.2014.
Unity Tutorials. 2014. Awake and Start. https://unity3d.com/learn/tutorials/modules/beginner/scripting/awake-and-start. 25.10.2014.
YoYo Games. 2011. Game Maker 8.1.141.
Warner Bros. Interactive Entertainment. 2013. Batman: Arkham Origins.
Waugh, R. 2014. Ten Classic Games That Changed the World: From Space Invaders to Destiny. https://uk.news.yahoo.com/ten-classic-games–how-space-war--angry-birds-and-call-of-duty-changed-history-133138507.html#485phHy. 11.11.2014.
Wikipedia. 2014a. Atari 2600. http://fi.wikipedia.org/wiki/Atari_2600. 12.11.2014.

Wikipedia. 2014b. iPhone 6. http://fi.wikipedia.org/wiki/IPhone_6. 12.11.2014.
Wikipedia. 2014c. Kiintolevy. http://fi.wikipedia.org/wiki/Kiintolevy. 12.11.2014.