Ringo Leidén

Verkkomoninpelin suorituskyvyn optimointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

27.4.2016

Metropolia

Tekijä Otsikko Sivumäärä Aika	Ringo Leidén Verkkomoninpelin suorituskyvyn optimointi 37 sivua 27.4.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Ryhmänjohtaja, akatemiatutkija Matias Palva Lehtori Antti Laiho

Insinöörityön tavoitteena oli parantaa projektipelin suorituskykyä käyttämällä siihen tarkoitettuja työkaluja. Projektipeli on ensimmäisen persoonan ammuskelupeli, joka on kehitetty neurotieteelliseen tutkimuskäyttöön. Työssä tutkittiin työkaluja, kuten Profiler, jolla suorituskyvyn pullonkaulat pystyttiin tunnistamaan. Optimointitekniikoista tutkittiin muun muassa 3D-mallien optimointia, tekstuurien ja äänitiedostojen pakkausta ja ohjelmointitapojen merkitystä suorituskykyyn. Peligrafiikan historiaa tutkittiin, jotta saatiin kuva optimoinnin tarpeesta ja grafiikan tärkeydestä pelin menestymisen kannalta.

Optimoinnin merkitys peleissä on suuri, koska monimutkaiset 3D-maailmat vaativat paljon laskentatehoa. Jotta pelille saadaan mahdollisimman laaja potentiaalinen käyttäjäkunta, tulee pelien toimia myös vanhemmalla laitteistolla. Tämän mahdollistavat optimointitekniikat, joilla saadaan laskettua pelin teknisiä vaatimuksia. Mobiililaitteille pelejä kehitettäessä vaaditaan usein optimointia, jotta peli toimii edes uusimmilla laitteilla.

Projektipeliin tehtiin muun muassa valokartat ja laskettiin occlusion culling, jolla piilotettiin ylimääräiset peliobjektit. Lisäksi kameran, valojen, laatutasojen ja pelimoottorin yleiset asetukset käytiin läpi, jotta suorituskykyä saatiin parannettua entisestään. Koodin optimoinnissa karsittiin turhia komentoja ja parannettiin muistinhallintaa. Jotkin optimointitekniikat, kuten object pooling, jäivät projektipeliin toteuttamatta.

Insinöörityön lopputuloksena projektipelin suorituskykyä saatiin parannettua huomattavasti ja yhden kuvan renderöinnin vaatimaa laskenta-aikaa saatiin laskettua prosessorilla noin 35 prosenttia ja näytönohjaimella jopa 75 prosenttia. Näin kuvataajuus saatiin nostettua yli tavoitekuvataajuuden 30 kuvaa sekunnissa. Kuvataajuuden alhaisuudesta johtunut pelikokemuksen katkonaisuus saatiin näin poistettua, mikä paransi pelikokemusta.

Avainsanat	optimointi, pelikehitys, 3D, Unity

Author Title Number of Pages Date	Ringo Leidén Performance optimization in an online multiplayer game 37 pages 27 April 2016
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Matias Palva, Group leader, Academy Research Fellow Antti Laiho, Senior Lecturer

Performance optimization is important in video games because complex 3D game worlds require large amounts of computing power. To reach the largest potential user base, games need to run also on lower end hardware. Optimization makes it possible to lower the hardware requirements of games, leading to a larger user base. In mobile game development, optimization is often required to make games run even on higher-end devices.

The goal of the final year project was to increase the performance of a multiplayer game by using suitable tools. The game is a first person shooter that is being developed for neurological research purposes. Tools such as Profiler were studied in order to identify the performance bottlenecks of the game. Optimization of 3D models and code, and compression of textures and audio files were also studied. The history of video game graphics was studied in order to understand the importance of optimization and graphics in video games.

To increase the performance of the project game, lightmaps were created and occlusion culling was calculated. Also settings for cameras, lights and the game engine were adjusted to increase performance. Code optimization concentrated on removing redundant code and making memory management better. Some techniques, like object pooling, could have been implemented to further improve the performance of the code of the game.

As a result of the final year project, the performance of the project game was improved remarkably, and the computing time required to render a frame was reduced by approximately 35% on the CPU and as much as 75% on the GPU. Thus the framerate of the game was increased above the goal framerate of 30 frames per second. This removed stuttering and improved overall gameplay.

Keywords	optimization, game development, 3D, Unity

Sisällys

Lyhenteet

1 Johdanto 1

2 Peligrafiikan ja -teknologian kehitys 3

2.1 Videopelien ensiaskeleet 3
2.2 Pelihallit ja kotikonsolit 4
2.3 Tietokoneiden aikakausi 7

3 Pelien suorituskyvyn optimoinnin osa-alueet 10

3.1 Pelien tekniset vaatimukset ja optimoinnin merkitys 10
3.2 3D-mallien optimointi ennen pelimoottoria 11
3.3 Tarkkuustasot ja valaistus 12
3.4 Ohjelmoinnin vaikutus suorituskykyyn 15
3.5 Tekstuurien ja äänitiedostojen pakkaus 16

4 Optimoinnin toteutus Unity-pelimoottorilla 17

4.1 Suorituskyvyn analysointi ja ongelmakohtien tunnistaminen 17
4.2 Kameran, valaistuksen ja pelimoottorin asetukset 19
4.3 Piirtokäskyt ja batching 25
4.4 Occlusion Culling ja tarkkuustasot 26
4.5 Koodin ja verkkoliikenteen optimointi 29
4.6 Tulokset 31

5 Yhteenveto 33

Lähteet 35

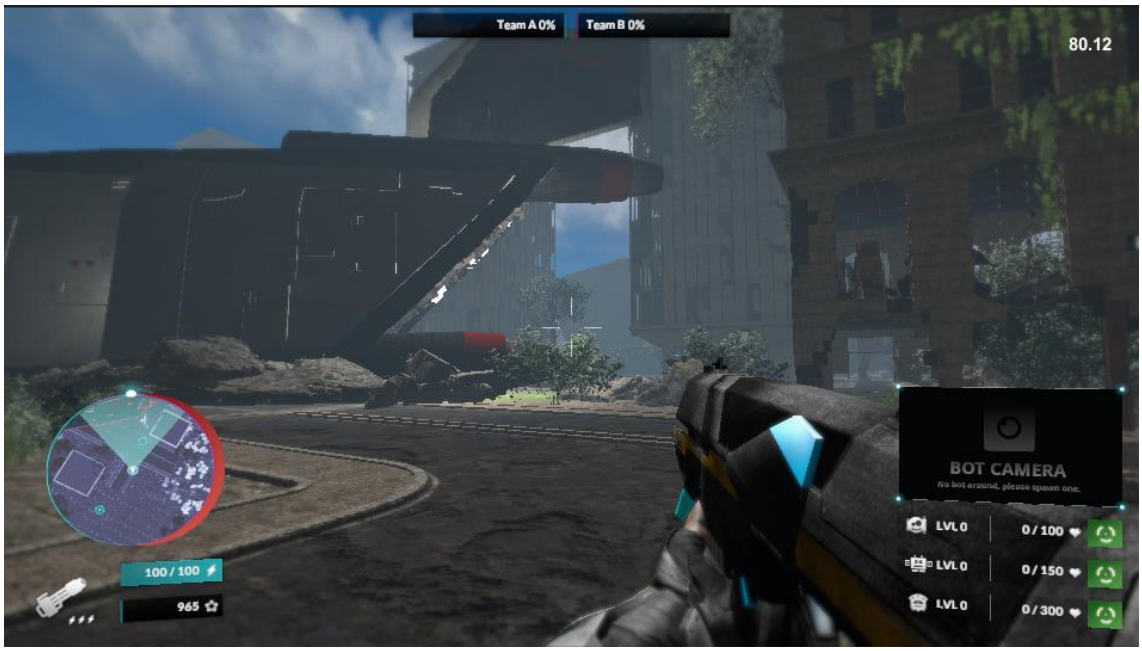
Lyhenteet

FPS	First Person Shooter. Peligenre, jossa pelihahmo kuvattu ensimmäisessä persoonassa.
FPS	Frames Per Second. Kuvataajuus: montako kuvaa sekunnissa näytölle renderöidään.
GI	Global Illumination. Tekniikka epäsuoran valaistuksen tuottamiseen peleissä.
LOD	Level of detail. 3D-mallin tarkkuustasot.
RGB	Red, Green and Blue. Värijärjestelmä, jossa käytössä punainen, vihreä ja sininen komponentti.

1 Johdanto

Hyvän pelin tulee toimia sujuvasti ja näyttää mahdollisimman hyvältä. Näiden vaatimusten yhteensovittaminen voi kuitenkin olla hankalaa, joten tarvitaan optimointia, jotta laitteiston suorituskyky saadaan parhaiten hyödynnettyä. Insinöörityön tarkoituksena on tutkia ja toteuttaa optimoinnin osa-alueita eri vaiheissa pelikehitysprosessia. Pääpaino on korkean tason optimoinnilla ja työkaluilla, joita nykypäivän pelimoottorit tarjoavat. Lisäksi tutkitaan peligrafiikan kehitystä, jotta saadaan kuva sen tärkeydestä pelikokemuksen kannalta.

Insinöörityö toteutetaan Helsingin yliopiston neurotieteen tutkimuskeskuksessa, osana peliprojektia. Projektissa toteutetaan Project:Neural-verkkomoninpeliä (kuva 1), joka on pelityypiltään First Person Shooter (FPS) eli ensimmäisen persoonan ammuskelu. Peliprojektin tavoitteena on auttaa neurotieteellisessä tutkimuksessa. Peli on vielä kehityksessä, joten kuvat ja tulokset eivät välttämättä täsmää lopullisen pelin kanssa.



Kuva 1. Project:Neural-pelin näkymä.

Project:Neural on peli, jolla tutkitaan uusien kykyjen oppimista ja sitä, miten aivot muuttuvat tämän oppimisen myötä. Peli myös antaa mahdollisuuden moniuloitteiseen kognitiivisen kyvykkyyden mittaamiseen, koska se sekä antaa visuomotorisia tarkkaavaisuus- ja päätöksentekohaasteita sekunti sekunnilta että vaatii pidempien aikavälien taktisia ja strategisia ratkaisuja. Peli antaakin tässä ekologisesti paljon validimman alustan kuin näitä ominaisuuksia erikseen mittaavat psykologiset testit. Project:Neural saattaa myös mahdollistaa kognitiivisten kykyjen kehittämisen tai niiden ikääntymisen tuoman heikkenemisen hidastamisen tai mahdollisesti jopa uusien hoitomuotojen kehittämisen useisiin aivosairauksiin.

Pelissä ohjataan scifi-tyyppistä soturia, jolla on energia-ase. Käytössä on myös ohjattavia robotteja, jotka voivat vahingoittaa vihollista. Pelissä on kehitysvaiheessa pelimuotoina kaksin- ja nelinpeli kahdessa joukkueessa. Pelialueet ovat laajoja, ja tavoitteena on korkea graafinen taso, joten optimointi on suuri osa kehitysprosessia. Koska peliä kehitetään tutkimuskäyttöön, pitää sen myös toimia tasaisesti ja luotettavasti ja hyvän pelikokemuksen pitää olla toistettavissa jokaiselle koehenkilölle.

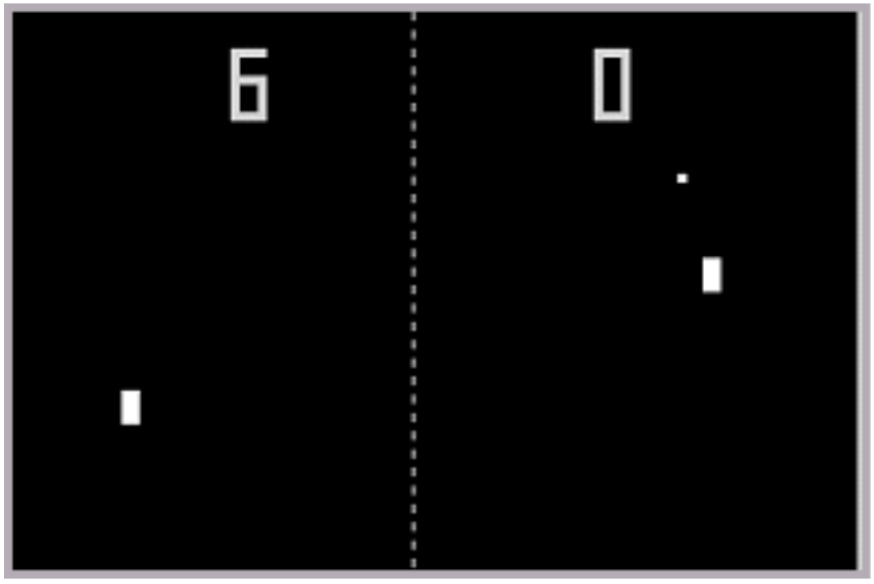
Päätavoitteena on parantaa projektipelin suorituskykyä tutkimalla optimointitekniikoita ja toteuttamalla niitä Unity-pelimoottorilla.

2 Peligrafiikan ja -teknologian kehitys

2.1 Videopelien ensiaskeleet

Ajatus tietokonepeleistä on ollut olemassa jo 1940-luvulla, ja muun muassa shakkia pelaavia tietokoneita ohjelmoitiin jo 1950-luvulla. Ensimmäisenä varsinaisena videopelinä pidetään kuitenkin Steve Russellin vuonna 1962 kehittämää Spacewar!-peliä, jossa kaksi pelaajaa taistelivat avaruusaluksilla vastakkain. [1.]

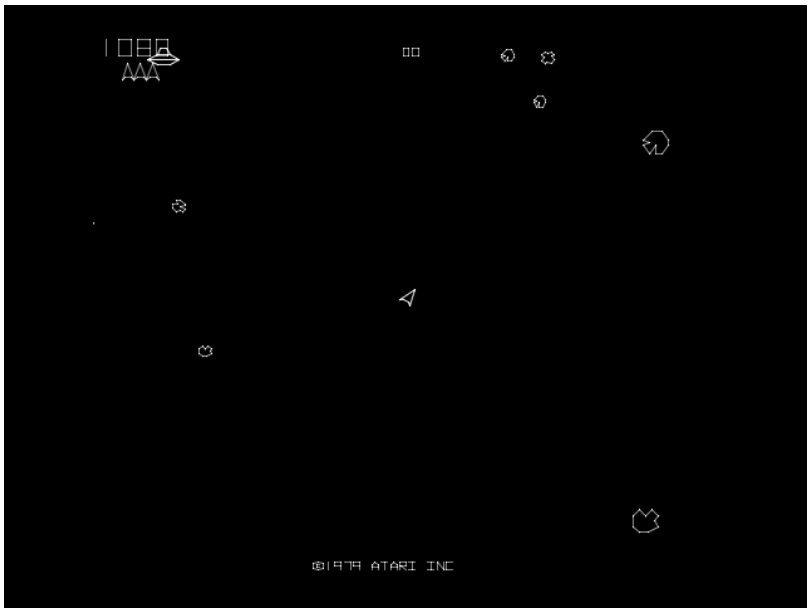
Ensimmäisissä videopeleissä grafiikka oli hyvin yksinkertaista ja laitteet oli rakennettu yhtä peliä varten. Näin oli myös ensimmäisissä kotikonsoleissa kuten Pong (kuva 2), jossa ohjattiin kahta valkoista palkkia mustavalkoisella ruudulla. Pong julkaistiin pelihallikoneena vuonna 1972 ja kotikonsolina vuonna 1975. Alkuaikojen peleihin värejä saatiin kalvoilla, jotka asetettiin television eteen. Tätä kalvotekniikkaa käytti jo ensimmäinen koteihin myyty pelikonsoli Magnavox Odyssey. Ensimmäisenä ”tietokonepelinä” pidetään Gunfightia (1975), sillä siinä käytettiin ensimmäistä kertaa mikroprosessoria puolijohdevirtapiirien sijasta. [2; 3.]



Kuva 2. Pong-peli vuodelta 1972 [3].

Videopelien grafiikan alkuaikoina kilpailevia näyttöteknologioita olivat rasterointi- ja vektorigrafiikka. Vektorigrafiikassa kuva piirretään ruudulle muokkaamalla elektroneita suoraan. Tässä tekniikassa ruudunpäivitystaajuus riippuu piirrettävien objektien määrästä,

joten se ei ole tasainen. Positiivisena puolena piirtojälki oli tasainen ja pyöreät muodot näyttivät yhtenäisiltä. Tunnetuimpia vektoritekniikalla tehtyjä pelejä on Asteroids (kuva 3), jossa avaruusalus tuhoaa asteroideja. [2; 4.]



Kuva 3. Asteroids, Atari 1979 [5].

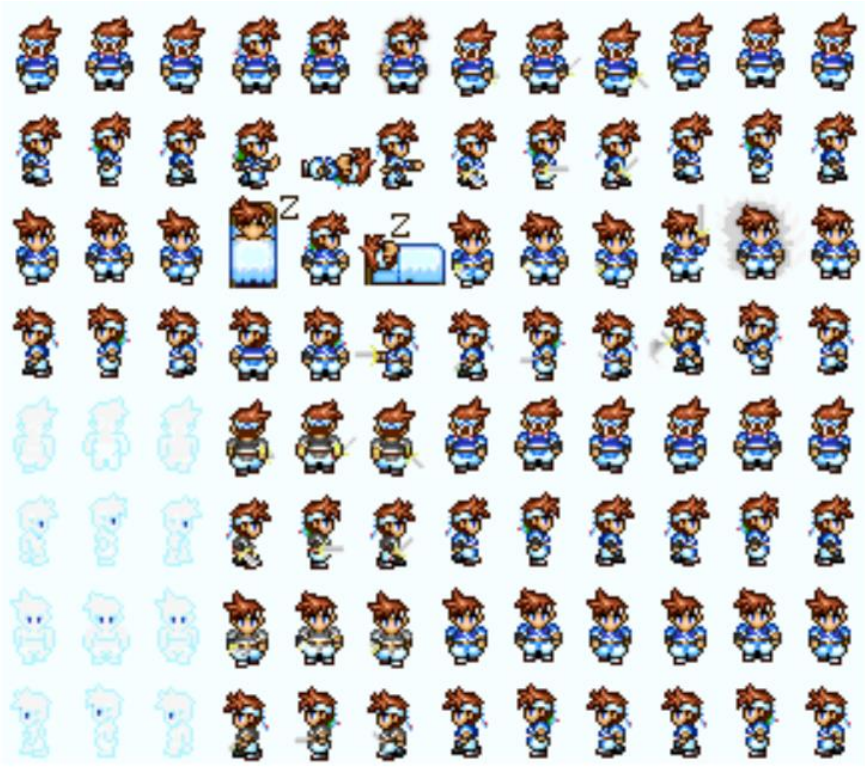
Rasteroinnissa kuva muodostetaan ruudulle rivi riviltä muokaten jokaisen rivin elektroneita. Tämä luo ruudukon, jonka jokainen solu on joko päällä tai pois. Nämä solut kehittyivät myöhemmin sisältämään RGB-arvoja. Tällä saadaan tasainen ruudunpäivitystaajuus, mutta koska kuva muodostuu neliöistä, pyöreät muodot näyttävät kulmikkailta. Rasterointitekniikan monipuolisuus ja suosio pelihallipeleissä vakiinnutti sen aseman. [2; 4.]

2.2 Pelihallit ja kotikonsolit

Ensimmäisten sukupolvien pelit olivat 2D-pelejä, eli toiminta tapahtui kahdessa ulottuvuudessa ja oli usein kuvattu ylhäältä tai sivulta katsoen. Myös kolmiulotteisuuteen pyrittiin jo alkuaikoina sellaisilla peleillä kuin Battlezone. Suuri askel kolmiulotteisuuteen olivat isometriset pelit. Nämä olivat pelejä, jotka matkivat 3D-maailmaa siirtämällä pelaajan katsontakulman yläviistoon. Vaikka pelit edelleen toimivat kahdessa ulottuvuudessa, sai pelaaja illuusion kolmiulotteisuudesta. Ensimmäinen isometrinen peli oli Segan vuonna 1982 julkaisema Zaxxon (kuva 4). [2.]

Kuva 4. Zaxxon, Sega 1982 [7].

Kolmiulotteisuutta matkittiin myös sprite scaling -tekniikalla. Spritet ovat kuvia, joita pystytään animoimaan ja liikuttamaan ruudulla (kuva 5). Esimerkiksi pelaajahahmo ja viholliset koostuvat usein spriteistä. Sprite scaling -tekniikassa näitä spritejä skaalataan sitä pienemmiksi, mitä kauempana ne olivat pelaajasta, jolloin syntyy illuusio perspektiivistä. Tätä tekniikkaa käytettiin useissa peleissä, kuten Turbo (Sega, 1981) ja Hang-On (Sega, 1985). [2.]



Kuva 5. Esimerkki pelihahmon spriteistä, joilla se pystytään animoimaan [3].

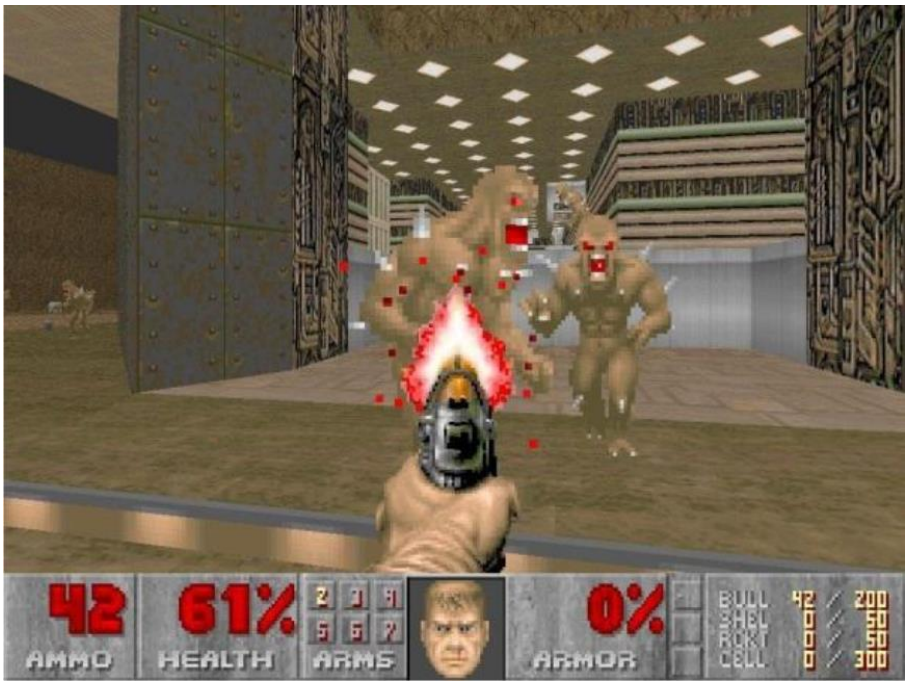
Yhdysvalloissa ja Euroopassa Atari johti pitkään kotikonsolimarkkinoita. Nintendo julkaisi Famicom-järjestelmän Japanissa vuonna 1983, ja Yhdysvalloissa ja Euroopassa konsoli julkaistiin nimellä Nintendo Entertainment System (NES) vuosina 1985 ja 1986. Super Mario Bros- ja Legend of Zelda -pelit varmistivat Nintendon paikan kotikonsolimarkkinoiden johtajana. Graafisesti tai laskentateholtaan kotikonsolit eivät pärjänneet pelihallikoneille, mutta kotona pelaamisen kätevyys osoittautui tärkeämmäksi kuin grafiikka. [3; 6.]

Seuraavan sukupolven konsolit, kuten Super Nintendo Entertainment System (SNES) ja Sega MegaDrive, nostivat peligrafiikan tasoa huomattavasti. Esimerkiksi samanaikaisten värien määrä näytöllä nousi SNES:llä 256:een edeltäjänsä 25 väristä. Myös samanaikaisten spritejen määrä kasvoi huomattavasti, joten peleistä oli mahdollista tehdä entistä näyttävämmän näköisiä. [6; 7.]

Myös kolmiulotteisuus peleissä mahdollistui tehokkaampien konsoleiden myötä, ja kolmiulotteisia polygoneja nähtiin esimerkiksi Star Fox -pelissä (Nintendo, 1993). Polygonit vaativat kuitenkin paljon laskentatehoa, eivätkä ne yleistyneet vielä 16-bittisessä konsolisukupolvessa. [2.]

2.3 Tietokoneiden aikakausi

Vaikka kotikonsolitkin kehittyvät jatkuvasti, olivat suurimmat graafiset edistysaskeleet nähtävissä ensin tietokoneilla. Kuvassa 6 näkyvä Doom (id Software, 1993) asetti riman korkealle kolmiulotteisten pelien saralla. Vaikka peli ei ollut aidosti kolmiulotteinen vaan käytti spritejä, sen luoma illuusio kolmiulotteisuudesta oli hyvin vahva. Doom-pelisarjan menestyksen myötä id Software julkaisi vuonna 1996 uuden pelinsä Quake, joka oli puolestaan aidosti kolmiulotteinen ja jonka koko pelimaailma koostui polygoneista. [2; 3.]



Kuva 6. Doom (id Software, 1993) [3].

Quake (kuva 7) merkitsi alkua uudelle peligrafiikan aikakaudelle, sillä se oli ensimmäinen peli, joka vaati erillisen grafiikkaprosessorin (GPU) toimiakseen. Nykytietokoneissa grafiikkaprosessori on itsestäänselvyys, mutta vielä 1990-luvun puolivälissä näin ei ollut. [2; 3.]

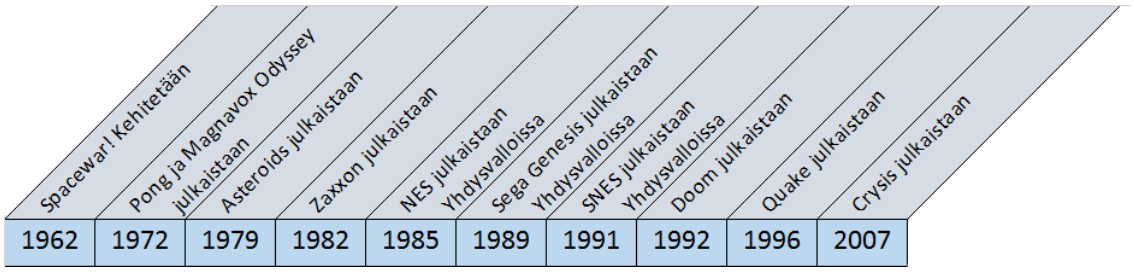
Kuva 7. Quake (id Software, 1996) [9].

Tietokoneiden yleistymisen jälkeen konsolit eivät pystyneet kilpailemaan suorituskyvyssä tietokoneiden kanssa. Tämä johti myös siihen, että pelien grafiikka on edelleen parhaimmillaan tehokkaalla tietokoneella. Konsolien etuna tosin on se, että niiden pelit optimoidaan tietylle laitteelle, jolloin kaikki käyttäjät saavat mahdollisimman hyvän grafiikan tason. [10, s. 285–287.]

Tietokonepeleissä käyttäjien tietokoneet voivat olla tehottomampia, jolloin grafiikka usein kärsii ensimmäisenä. Tietokoneiden vaihtelevuuden vuoksi PC-pelikehittäjät ovat joutuneet usein valitsemaan parhaan grafiikan ja mahdollisimman laajan yleisön välillä. Esimerkiksi Crysis (kuva 8) asetti uuden riman peligrafiikalle, mutta peli toimi vain kaikkein uusimmilla tietokoneilla. [2; 4.]

Kuva 8. Crysisin (Crytek, 2007) vaikuttava grafiikka [11].

Tasapainoilu grafiikan tason ja suorituskyvyn välillä on tärkeä osa pelikehitystä, ja siinä auttaa hyvä optimointi. Pelialan historia on suhteellisen lyhyt (kuva 9), mutta alusta asti grafiikka on ollut tärkeä osa pelikokemusta. Pelien alkuaikoina resurssit olivat hyvin vähissä ja tarkka suunnittelu ja optimointi oli edellytys toimivan pelin kehittämiseen.



Kuva 9. Peligrafiikan ja -teknologian kehityksen aikajana.

Konsoleissa optimointi keskittyy käyttämään konsolin suorituskyvyn mahdollisimman tehokkaasti. Tietokoneissa puolestaan usein tähdätään monelle eritasoiselle tietokoneelle, jolloin hyvä optimointi auttaa varmistamaan pelikokemuksen laadun kaikilla käyttäjillä.

3 Pelien suorituskyvyn optimoinnin osa-alueet

3.1 Pelien tekniset vaatimukset ja optimoinnin merkitys

Kun halutaan luoda kilpailukykyinen peli, sen tulee näyttää mahdollisimman hyvältä. Hyvällä optimoinnilla voidaan vähentää laitteiston kuormaa, jolloin esimerkiksi graafisia ominaisuuksia voidaan lisätä. Pelejä kehitettäessä on oltava tieto kohdealustoista eli siitä, millaisella laitteistolla peliä pitäisi pystyä pelaamaan sujuvasti. Tietokoneelle kehitettäessä määritetään usein minimilaitteisto, jolla pelin pitäisi toimia. Kun peli optimoidaan toimimaan sujuvasti niin sanotusti huonolla laitteistolla, voidaan olettaa, että se toimii paremmin keskivertolaitteistolla. [10, s. 1; 15; 24.]

Pelien suorituskykyä tarkasteltaessa tärkeimmät tietokoneen resurssit ovat prosessori, näytönohjain ja muisti. Onkin tärkeää saada pelin vaatimat resurssit tasapainotettua näiden välillä, jotta kaikkia pystytään hyödyntämään tehokkaasti (10, s. 12). Tavoitteena on saada kuvataajuus (framerate, frames per second, FPS) mahdollisimman tasaiseksi ja tavoitteiden mukaiseksi. Kuvataajuus kertoo, kuinka monta kuvaa sekunnissa pystytään näytölle muodostamaan. Tätä kuvan muodostusta kutsutaan renderöinniksi (rendering).

Peleissä kuvataajuus yritetään usein pitää välillä 30–60 kuvaa sekunnissa. Jos kuvataajuus on 30, on pelillä 33 ms aikaa laskea ja renderöidä seuraava kuva. Peleissä laskenta tapahtuu ohjelmointisilmukoissa (loop), joita toistetaan mahdollisimman usein. [12, s. 81.]

Näytönohjaimessa suorituskykyä rajoittaa usein sen muistin kaistanleveys. Mitä vähemmän kaistaa piirrettävä grafiikka vie näytönohjaimelta, sitä nopeammin kuvaa pystytään näytölle renderöimään, jolloin kuvataajuus kasvaa.

Prosessorin kuormitukseen vaikuttaa eniten renderöitävien objektien määrä ja niihin vaikuttavat valot ja varjot. Kun näytölle tulee piirtää objekti, pelimoottori antaa piirtokäskyn (draw call), joka sisältää tietoa esineestä ja sen ominaisuuksista. Jos näitä piirtokäskyjä tulee liikaa, laskee se pelin suorituskykyä. [13, s. 207.]

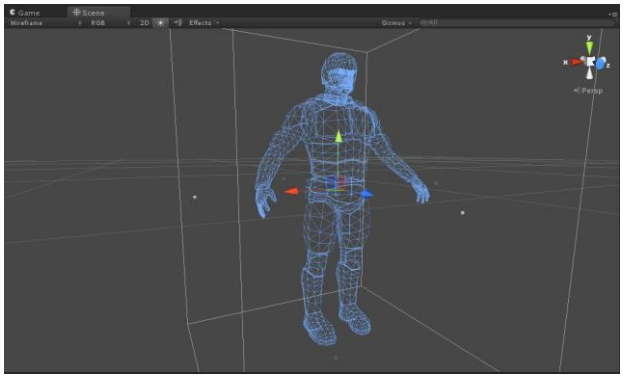
Jo suunnitteluvaiheessa on tärkeää määritellä tarkasti alustat, joille peli julkaistaan. Mobiililaitteissa on huomattavasti vähemmän laskentatehoa kuin kotitietokoneissa, joten graafikoiden, suunnittelijoiden ja ohjelmoijien on tiedettävä alusta asti pelin kohdealusta. Käytettävissä oleva laskentateho rajoittaa sitä, mikä on mahdollista niin graafisesti kuin pelimekaniikan puolesta. [14.]

Mobiililaitteissa on laskentatehon lisäksi otettava huomioon tallennustila. Liian suuri peli ei mahdu kaikille laitteille, ja se on hitaampi ladata laitteeseen. Tallennustilaa vievät yleensä eniten grafiikka ja kolmiulotteiset ympäristöt, joten mobiilipeleissä harvoin näkee laajoja kolmiulotteisia maailmoja, joissa on realistinen grafiikka.

Kohdealusta vaikuttaa myös suuresti pelimekaniikkojen suunniteluun, ja optimoinnin kannalta tärkeää on etenkin muistinhallinta. Mobiililaitteilla käyttömuistia on usein vähemmän, joten pienetkin muistivuodot (ks. 3.4 Ohjelmoinnin vaikutus suorituskykyyn) saattavat nopeasti kaataa koko pelin tai jopa laitteen.

3.2 3D-mallien optimointi ennen pelimoottoria

Monimutkaiset 3D-mallit tuotetaan niiden tekemiseen suunnitelluilla 3D-mallinnusohjelmilla. 3D-mallinnuksessa kohteesta luodaan mesh (kuva 10) eli verkko, joka koostuu toisiinsa liitetyistä monikulmioista eli polygoneista. Primitiivisin polygoni on kolmio (triangle), joka puolestaan koostuu vertekseistä (vertex) eli pisteistä 3D-avaruudessa. Yleensä viimeistään pelimoottori muuttaa kaikki monikulmiot kolmioiksi. Tämä muutos tulisi kuitenkin tehdä jo mallinnusvaiheessa, sillä monimutkaiset muodot aiheuttavat turhia kustannuksia pelimoottorille. [15; 16.]



Kuva 10. Projektipelin pelaajamallin mesh eli polygoniverkko.

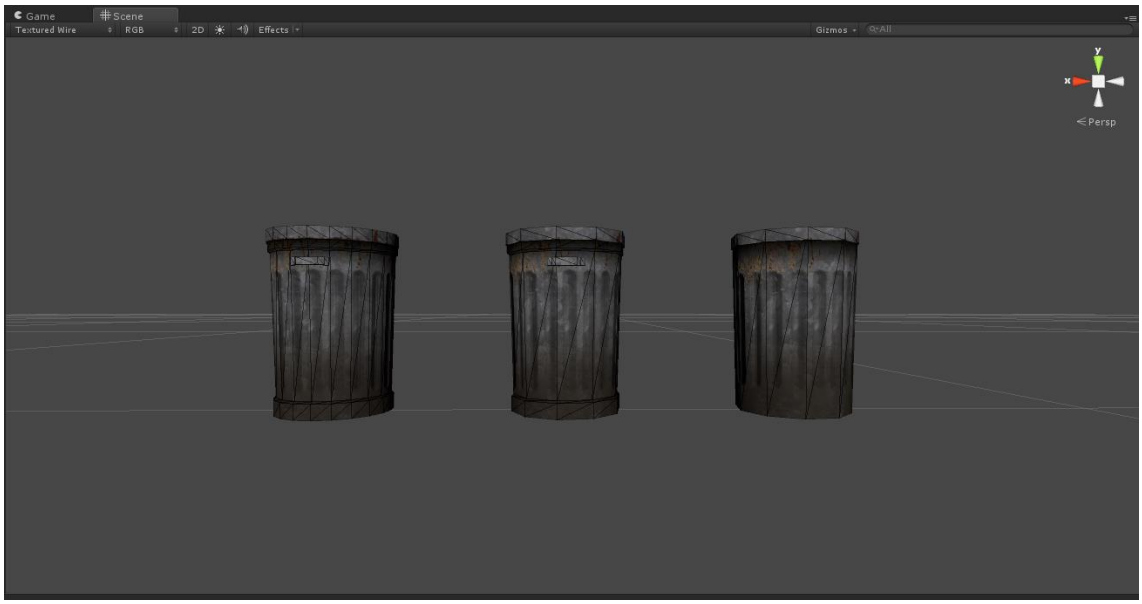
Malleja luotaessa on tiedettävä niiden käyttötarkoitus, sillä peleissä vaaditaan paljon kevyempiä malleja kuin esimerkiksi animaatioelokuvissa. Videotuotannossa jokaisen kuvan renderöintiin voidaan käyttää useita tunteja, kun taas peleissä renderöinnin pitää tapahtua ”reaaliajassa”. 3D-mallien monimutkaisuus voi aiheuttaa suuria ongelmia pelien suorituskyvyssä. 3D-malleja tulisikin optimoida jo ennen pelikehitysympäristöön tuomista. Polygonien lukumäärä tulisi pitää mahdollisimman pienenä, sillä se vaikuttaa renderöintiin tarvittavan ajan määrään. Yksittäisten mallien suuri polygonimäärää aiheuttaa usein suurempaa rasitusta näytönohjaimelle kuin prosessorille. Prosessori tekee muun muassa valaistukseen liittyvää laskentaa peliobjektikohtaisesti, joten prosessorin kannalta on tärkeämpää pitää renderöitävien peliobjektien määrä pienenä. Koko pelin verteksimäärän tulisi mobiililaitteilla olla alle 100 000 ja tietokoneilla enintään muutama miljoona. [15; 16.]

Myös turhat yksityiskohdat tulisi pyrkiä karsimaan peleihin suunnitelluista malleista. Esimerkiksi pinnan muodon yksityiskohtia voidaan jättää mallintamatta ja illuusio pinnan tekstuurista luoda Normal- ja Bump Map -tekstuureilla. Nämä tekstuurit sisältävä joko suunta- tai intensiteettitiedot esineestä, ja kun ne yhdistetään kuvatekstuuriin, saadaan litteään pintaan muodon illuusio. [17.]

Tietysti myös 3D-mallin käyttötarkoitus tulee ottaa huomioon. Esimerkiksi pelaajamalliin (joka on koko ajan lähellä kameraa) voidaan mallintaa enemmän yksityiskohtia, jotta se saadaan näyttämään mahdollisimman hyvältä.

3.3 Tarkkuustasot ja valaistus

Monimutkaisista 3D-malleista voidaan luoda yksinkertaisempia versioita, joita käytetään parantamaan suorituskykyä. Näitä versioita kutsutaan mallin tarkkuus- tai LOD-tasoiksi (Level Of Detail). Kuvassa 11 nähdään saman tynnyrimallin kolme eri tarkkuustasoa. Läheltä katsottaessa huomaa, kuinka yksityiskohtia puuttuu oikeanpuolimmaisesta tynnyristä, mutta kaukaa katsoessa eroja on vaikea nähdä, sillä tynnyreiden siluetti on sama. [18.]

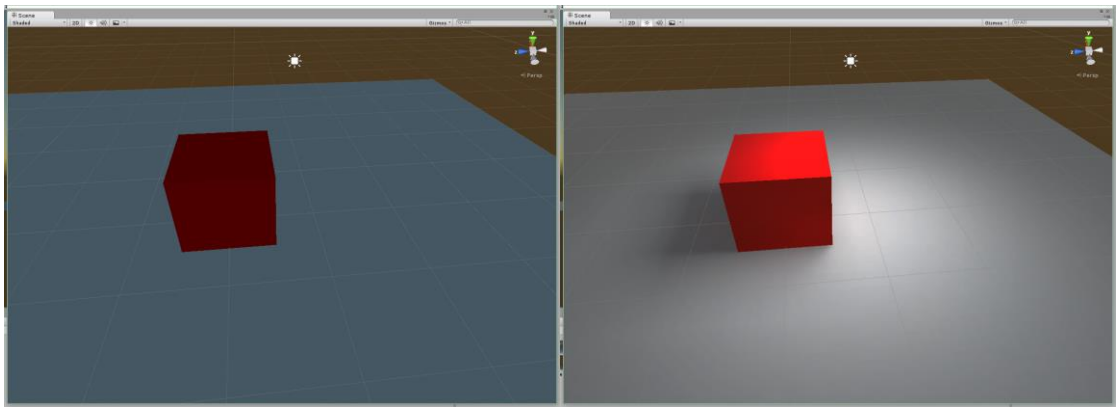


Kuva 11. Kolme tarkkuustasoa roskakorin 3D-mallista Unity-pelimoottorissa.

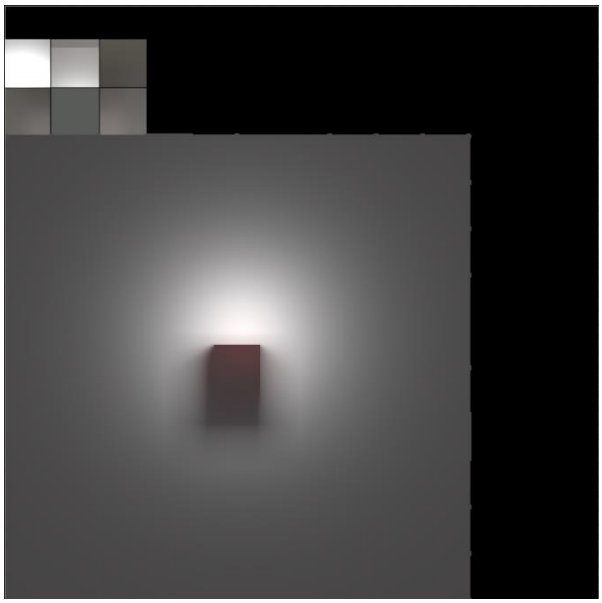
Tätä tekniikkaa käytetäänkin säästämään tietokoneen laskentatehoa, kun 3D-mallit ovat kaukana. Mitä kauempana objekti on, sitä yksinkertaisempia malleja voidaan käyttää ilman, että ihmissilmä huomaa eroa. Tärkeää on kuitenkin, että mallien siluetti pysyy mahdollisimman samana, koska muuten pelaaja voi huomata mallin vaihtumisen liikkuessaan lähemmäksi objektia. [18.]

Hyvin suunniteltu ja toteutettu valaistus on olennainen osa nykypäivän peliä. Valojen ja varjojen reaaliaikainen laskenta vie kuitenkin paljon resursseja, joten usein suurin osa valoista lasketaan etukäteen. Tästä prosessista saadaan niin sanottuja valokarttoja (lightmaps), jotka kertovat, kuinka valot vaikuttavat pelitason objekteihin. Nämä valokartat yhdistetään pelattaessa objektien tekstuureihin, jolloin saadaan illuusio valojen vaikutuksesta. Tämä tekniikkaa toimii kuitenkin vain objekteilla, jotka eivät liiku pelin aikana, sillä valokartat luodaan jo kehitysvaiheessa. [19.]

Kuvassa 12 nähdään valokartan vaikutus yksinkertaiseen objektiin. Vasemmalla kuutio on varjossa, sillä valokarttoja ei ole vielä laskettu. Oikealla laskenta on tehty ja kuutioon vaikuttava valo ja sen varjo näkyvät selvästi. Tämän tilanteen valokartta nähdään kuvassa 13.



Kuva 12. Valokartan vaikutus.



Kuva 13. Esimerkki valokartasta.

Valokarttojen koko ja resoluutio vaikuttavat lopputulokseen, ja vaikka valokartat usein auttavat suorituskykyä, voivat liian tarkat valokartat myös hidastaa peliä ja viedä merkittävästi tallennuskapasiteettia. Tämän helpottamiseksi voidaan valokarttojen laatua laskea paremman suorituskyvyn saamiseksi. Useita valokarttoja voidaan myös pakata yhteen suorituskyvyn parantamiseksi. [19.]

3.4 Ohjelmoinnin vaikutus suorituskykyyn

Koodin optimoinnista puhuttaessa tulee usein esiin sanonta ”ennenaikainen optimointi on kaiken pahan alku ja juuri”. Tämä kuitenkin usein ymmärretään väärin, ja siksi optimointi unohdetaan kokonaan ennen projektin loppua. [20.]

Huonosti suunniteltu ja toteutettu koodipohja voi lamauttaa pelin tai ohjelmiston suorituskyvyn. Projektin alkuvaiheissa tehdyt virheet voivat myös olla lähes mahdottomia tai vähintään aikaa vieviä korjata projektin lopussa. On tärkeää osata havaita mahdolliset koodin suorituskyvyn pullonkaulat jokaisessa kehitysvaiheessa. Ennenaikaisella optimoinnilla tarkoitetaankin usein koodin pieniä tehottomuuksia, joiden korjaaminen ei alkuvaiheessa kannata suhteessa sen viemään aikaan. [20.]

Tietokoneissa käyttömuisti jakautuu kahteen tyyppiin, jotka ovat pino (stack) ja keko (heap). Pinomuistia varataan ja vapautetaan jatkuvasti. Sinne tallennetaan muun muassa paikalliset muuttujat, jotka katoavat heti, kun funktiosta poistutaan. Pinomuistin varaaminen ja vapauttaminen on automaattista ja nopeaa. [21.]

Kekomuisti puolestaan on pysyvää, ja sinne tallennettu tieto (esim. globaalit muuttujat) ei vapaudu itsestään. Jos ohjelma varaa jatkuvasti kekomuistia mutta ei koskaan vapauta sitä, syntyy niin sanottu muistivuoto (memory leak). Muistivuodot ovat hyvin haitallisia, sillä ne varaavat muistia, kunnes se loppuu. Tämä johtaa yleensä ohjelman hidastumiseen ja lopulta kaatumiseen. [21.]

Muistinhallinta vaihtelee suuresti ohjelmointikielittäin. Monet pelinkehitysohjelmat, kuten Unity, tarjoavat automaattisen muistinhallinnan. Tämä tarkoittaa, että ohjelma varaa ja vapauttaa kekomuistia automaattisesti, jolloin muistivuotojen riski vähenee huomattavasti. [22.]

Vaikka automaattinen muistinhallinta olisi käytössä, voi huonoilla ohjelmointitavoilla silti varata liikaa muistia, jolloin ohjelma hidastuu turhaan. Jos esimerkiksi muuttuja luodaan jokaisessa ohjelmointisilmukassa uudestaan, se vaatii aina muistin varaamista ja vapauttamista. Kannattaa siis luoda globaali muuttuja, jos sen arvoa muokataan usein. Näin muuttujan varaama muistipaikka pysyy samana eikä muistinhallinta joudu varaamaan ja vapauttamaan paikkaa uudestaan kymmeniä kertoja sekunnissa. [22.]

3.5 Tekstuurien ja äänitiedostojen pakkaus

Pakkaamalla kuva- ja äänitiedostot voidaan säästää huomattavasti muistia ja tallennustilaa (mikä on erittäin tärkeää etenkin mobiilipeleissä).

Peleissä kuvatiedostoja voi olla jopa tuhansia, jolloin on hyvin tärkeää, että ne on pakattu oikein. Jo ennen pakkaamista tulee miettiä, minkä kokoisena kuva tarvitaan. Jos kuva ei koskaan tule peittämään suurta pinta-alaa näytöstä, on sen resoluutiota turha pitää liian suurena. Esimerkiksi valikoiden kuvakkeet ovat aina samankokoisia, joten niiden resoluutio voidaan määrittää jo suunnitteluvaiheessa.

Sekä kuva- että musiikkitiedostoja voidaan pakata häviöttömästi (lossless) tai häviöllisesti (lossy). Häviötön pakkaus tarkoittaa, että tiedostosta ei poisteta mitään informaatiota. Hyvä esimerkki häviöttömästä pakkaamisesta ovat zip-tiedostot. Niihin voidaan pakata mitä tahansa tiedostoja, ja ne saadaan purettaessa takaisin täysin samanlaisina. Häviöttömiä pakkausmuotoja ovat muun muassa TIFF (kuva) ja Wav (ääni). [23.]

Peleissä käytetään usein kuitenkin häviöllisiä pakkausmenetelmiä. Vaikka informaatiota katoaa aina häviöllisissä menetelmissä, ne ovat huomattavasti tehokkaampia kuin häviöttömät menetelmät. Lisäksi sekä kuvista että äänitiedostoista voidaan poistaa paljon informaatiota, jota ihminen ei pysty havaitsemaan (esimerkiksi äänitaajuuksia, joita ihmiskorva ei kuule). Yleisimpiä häviöllisiä pakkausmuotoja ovat muun muassa PNG ja JPG (kuva) sekä MP3 ja OGG (ääni). [23.]

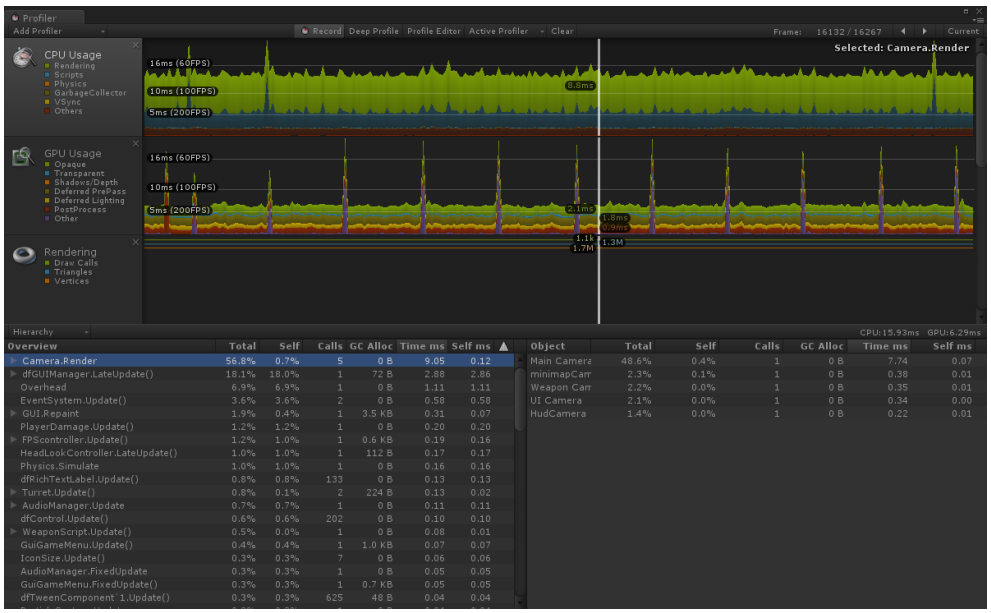
Äänitiedostoissa voidaan myös säästää paljon tallennustilaa pakkaamalla ne. Äänien toistaminen vaatii kuitenkin pakkauksen purkamista, joten jos äänitiedosto on hyvin lyhyt ja sitä soitetaan usein (esim. laukaus), voi olla parempi jättää se natiivimuotoon (yleensä wav). Tämä vie enemmän tallennustilaa, mutta säästää pakkauksen purkamiseen vaadittavat resurssit pelin aikana.

4 Optimoinnin toteutus Unity-pelimoottorilla

4.1 Suorituskyvyn analysointi ja ongelmakohtien tunnistaminen

Pelien suorituskyvyn optimointia aloitettaessa on tärkeää osata keskittyä oikeisiin osa-alueisiin. On osattava löytää suorituskyvyn ongelmakohdat ja pullonkaulat, jotta optimoinnista on hyötyä. Jos esimerkiksi näytönohjaimen resurssit eivät riitä, ei prosessoritehon vapauttaminen juuri vaikuta kuvataajuuteen.

Unityssä tärkein työkalu suorituskyvyn analysointiin on Profiler (kuva 14), joka kertoo hyvin tarkasti, mitkä osa-alueet vievät mitäkin resursseja. Peliä kehittäessä kannattaakin tutkailla Profileria ajoittain, jotta ongelmakohdat voidaan ratkaista hyvissä ajoin.



Kuva 14. Unityn Profiler-työkalu pelin suorituskyvyn analysointiin.

Kuvassa 14 tarkastellaan Prosessorin käyttöä (CPU Usage) ja nähdään, että kameroiden renderöiminen (kuvassa rivi Camera.Render) vie 56,8 % prosessorin kokonaiskulutuksesta ja kestää 9,05 millisekuntia. Kuvan oikeassa alalaidassa nähdään myös erittely kaikista pelitason kameroista ja niiden osuus kokonaiskulutuksesta. Nämä ovat suhteellisen normaaleja lukuja, ja kuvanottohetkellä kuvataajuus oli noin 60 kuvaa/s.

Profiler kertoo myös erillisten ohjelmointitiedostojen viemän tehon. Usein koodin vaatima suorituskyky on pientä verrattuna esimerkiksi renderöintiin, joten jos joku tiedosto vie

paljon laskentatehoa, kannattaa tutkia, voisiko sitä optimoida. Kuvasta 14 voidaan nähdä, että dfGUIManager-tiedoston LateUpdate-funktio vie 18,1 % prosessorin laskenta-ajasta, joten sen suorituskyvyssä on todennäköisesti parantamisen varaa.

Yleensä hitaassa koodissa kutsutaan liian usein raskaita, esimerkiksi renderöintiin vaikuttavia funktioita. Myös suoranaiset virheet (Error) koodissa voivat jäädä huomaamatta ja laskevat usein suorituskykyä huomattavasti.

Peliä kehittäessä on myös hyvä pitää editorin Statistics-ikkuna (kuva 15) avoinna. Siitä näkyvät muun muassa kuvataajuus (FPS), piirtokäskyt (Draw Calls) ja renderöitävien kolmioiden määrä (Tris). Nämä tiedot löytyvät myös Profilerista, mutta Statistics-ikkunaa on helpompi tarkkailla kesken yleisen pelikehityksen.

Kuva 15. Unityn Statistics-ikkuna.

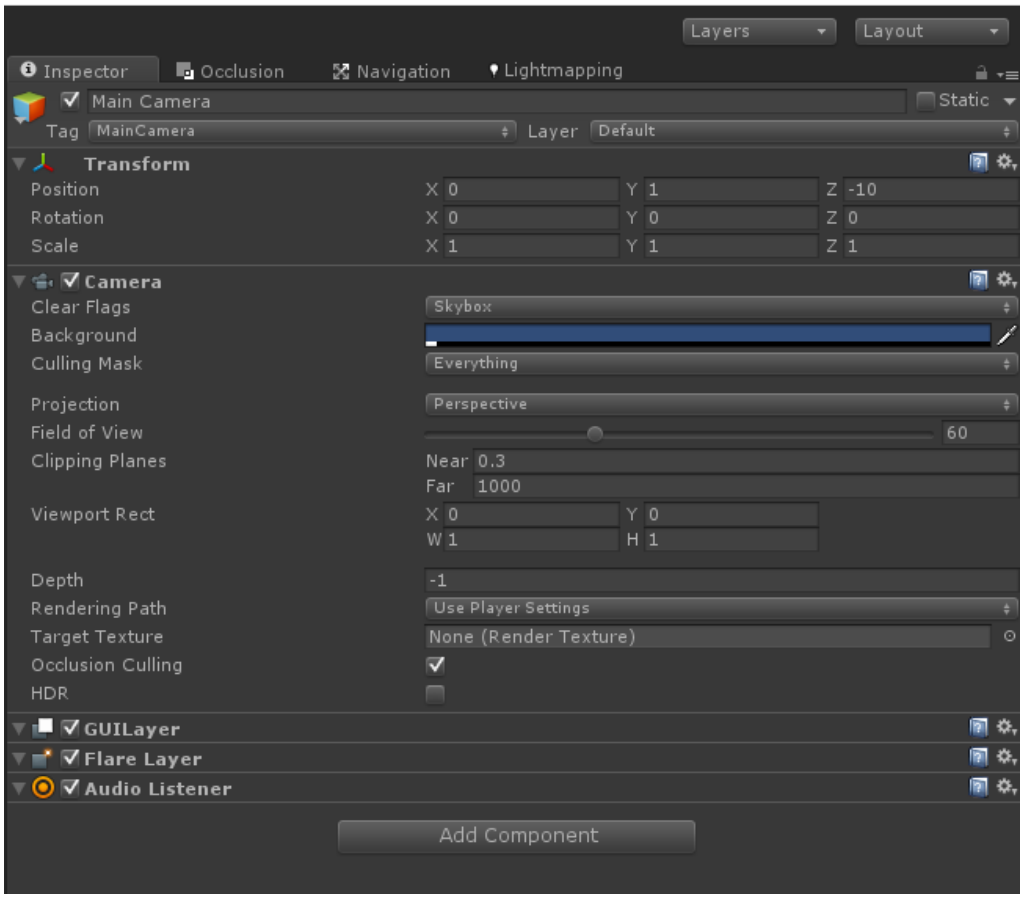
Projektipelissä pelitaso oli monimutkainen ja renderöitäviä objekteja oli paljon. Profiler ja Statistics-ikkuna olivat tärkeitä ongelmakohtien tunnistamiseen, jotta pelin kuvataajuus saatiin tavoitetasolle (ks. luku 4.6 Tulokset).

4.2 Kameran, valaistuksen ja pelimoottorin asetukset

Unityssä pelimaailma koostu peliobjekteista (GameObject). Kamera-peliobjekteilla määritetään, mitä näytölle renderöidään eli mitkä peliobjektit ovat samaan aikaan näytöllä. Kamera liitetään usein pelaajan ohjaamaan hahmoon, ja näin voidaan tutkia pelimaailmaa. Kameran asetuksilla voidaan vaikuttaa merkittävästi pelin suorituskykyyn.

Optimoinnin kannalta tärkeimmät kameran asetukset (kuva 16) ovat seuraavat:

- Culling Mask: Määrittää, millä tasoilla (Layer) olevat esineet renderöidään.
- Field of View: Kameran kuva-alueen laajuus asteina.
- Clipping Planes: Etäisyydet, joilla kamera aloittaa (Near) ja lopettaa (Far) renderöinnin. Arvot Unity-yksikköinä.



Kuva 16. Kameran asetukset Unity-pelimoottorissa.

Field of View -arvoa pienentämällä kuva-alue pienenee, jolloin renderöidään vähemmän. Tosin liian pienet tai suuret arvot alkavat nopeasti näyttää epärealistisilta. Jos pelissä on

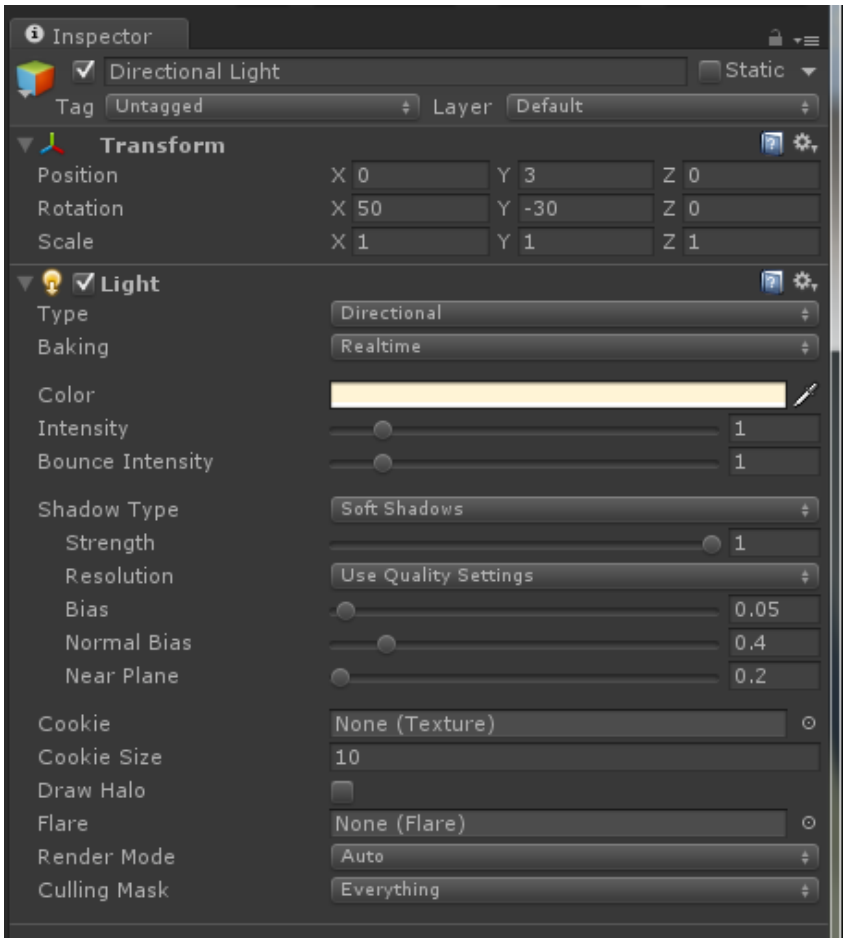
paljon näköesteitä (sumua, rakennelmia), voi Clippin Planesin Far-arvoa pienentää, jolloin kamera ei renderöi kaukana olevia objekteja. Jos taas peli on hyvin laaja ja avara, voi arvoa joutua suurentamaan, jotta maasto ei yhtäkkiä lopu.

Valojen merkitys optimoinnin kannalta on erittäin suuri. Etenkin varjojen laatu ja määrä vaikuttavat renderöinnin vaatimaan laskentatehoon. Onkin tärkeää suunnitella valojen käyttö tarkkaan, jotta saadaan tasapaino näyttävyyden ja suorituskyvyn välille. Valot renderöidään Unityssä joko verteksi- tai pikselivaloina. Näistä verteksivalot ovat huomattavasti nopeampia renderöidä, mutta niillä ei saada yhtä tarkkaa lopputulosta. [24.]

Valot ja varjot voivat olla Unityssä joko reaaliaikaisia tai valokarttoina. Reaaliaikaisia valoja tulee käyttää hyvin säästeliäästi, sillä ne vaativat erittäin paljon laskentatehoa. Ennalta tehdyt valokartat tekevät valoista jopa 2–3 kertaa tehokkaammat suorituskyvyn kannalta. [16; 24.]

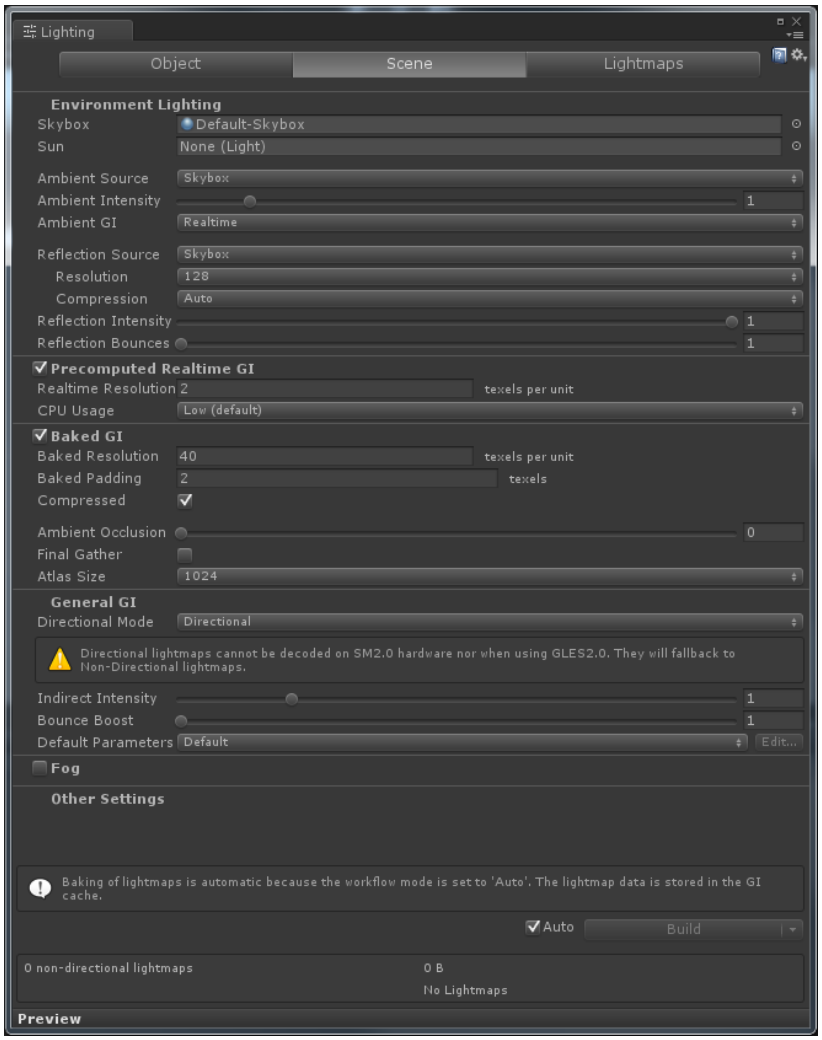
Optimoinnin kannalta tärkeimmät valojen asetukset (kuva 17) ovat seuraavat:

- Type (tyyppi): Määrittää valon tyypin.
- Baking: Määrittää, onko valo reaaliaikanen, valokarttana vai niiden sekoitus.
- Intensity ja Bounce Intensity: Määrittävät valon ja sen kimmokkeiden voimakkuuden. Mitä suurempi arvo, sitä enemmän laskentatehoa vaaditaan.
- Shadow type: Määrittää valon tuottamien varjojen tyypin. Vaihtoehtoina no shadows (ei varjoja), soft shadows (pehmeät varjot) ja hard shadows (kovat varjot). Pehmeät varjot ovat realistisin vaihtoehto, mutta vievät myös eniten laskentatehoa.
- Resolution: Varjojen laatuasetus. Yleensä kannattaa valita kohta ”Use Player Settings”, jolloin käytetään laatuasetuksissa määritettyjä arvoja.
- Render Mode: Valojen tärkeys renderöinnissä. Automaattinen vaihtoehto (auto) on usein paras, sillä se säätää reaaliajassa valon tehokkuutta ja laatua.
- Culling Mask: Määrittää, minkä tason (layer) objekteihin valo osuu. Poistamalla turhat kohteet voidaan säästää laskentatehoa.



Kuva 17. Valojen asetukset Unity-pelimoottorissa.

Valokarttojen ja yleisen valaistuksen asetukset löytyvät Lighting-ikkunasta (kuva 18). Sieltä voidaan säätää epäsuoraa valaistusta (Global Illumination), joka ei tule valoista. Epäsuora valaistus simuloi tapaa, jolla valo heijastuu esineistä. Tämä mahdollistaa esimerkiksi tekstuureissa olevien värien heijastumisen toisiin lähellä oleviin peliobjekteihin. Rajoituksena on kuitenkin, että epäsuora valaistus vaikuttaa vain peliobjekteihin, jotka eivät liiku. [25.]



Kuva 18. Valaistuksen asetukset Unity-pelimoottorissa.

Epäsuora valaistus voi olla ennalta laskettu (Baked GI), tai sitä voidaan muokata osittain reaaliaikaisesti (Precomputed Realtime GI). Valaistuksen asetuksissa (kuva 18) voidaan säätää muun muassa resoluutiota ja heijastusten määrää, jotka vaikuttavat vaadittuun laskentatehoon.

Unity-pelimoottorin asetuksissa on useita kohtia, joissa määritetään optimoinnin kannalta tärkeitä asetuksia. Monet niistä ovat oletusarvoisesti oikein, mutta jos suorituskyvyn kanssa on ongelmia, on asetukset hyvä käydä läpi.

Laatuasetuksista (Quality Settings, kuva 19) voidaan määritellä erilaiset tarkkuustasot, jotka ovat valittavissa, kun pelin käynnistää. Ne ovat myös hyvä työkalu optimoinnissa, sillä niillä voi nopeasti testata esimerkiksi varjojen vaikutusta suorituskykyyn.

Kuva 19. Unityn laatuasetukset.

Optimoinnin kannalta tärkeimmät laatuasetukset ovat seuraavat:

- Pixel Light Count: Samanaikaisten pikselivalojen maksimimäärä. Jos kamera näkee enemmän kuin tässä määritellyn määrän pikselivaloja, jätetään osa renderöimättä.

- Texture Quality: Voidaan asettaa yleinen tekstuureiden maksimiresoluutio. Voidaan määrittää täydestä resoluutiosta aina 1/8-resoluutioon saakka.

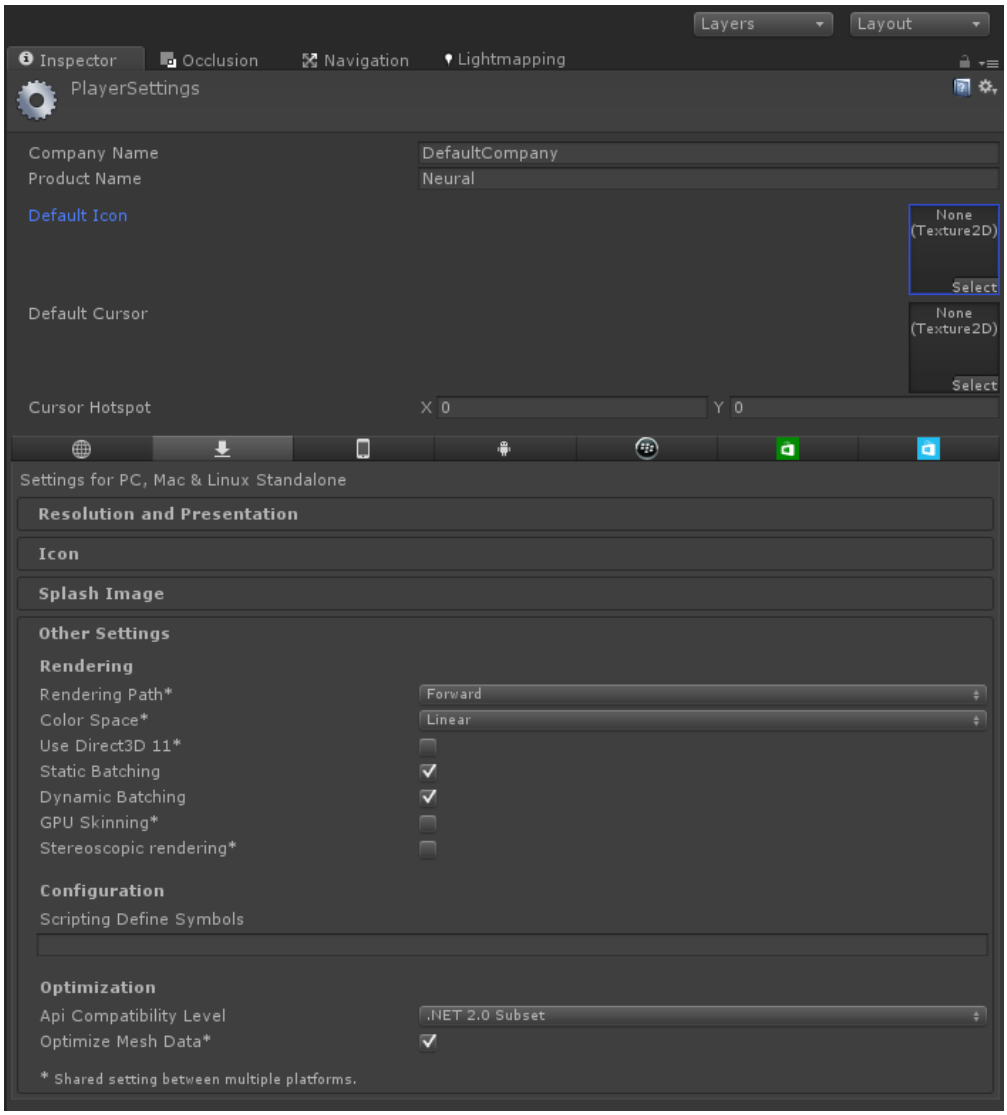
- Shadows: Määrittää mahdollisten varjojen tyypin. Tästä voidaan myös poistaa varjot kokonaan.

- Shadow Distance: Varjojen piirtoetäisyys. Jos varjot vievät huomattavasti laskentatehoa, kannattaa usein koettaa pienentää tätä arvoa.

- Maximum LODLevel: Mallien suurin mahdollinen tarkkuustaso. Esimerkiksi mobiililaitteilla ei välttämättä edes lähellä haluta näyttää mallien tarkinta versiota. Yleensä 0-taso on tarkin mahdollinen.

Lisäksi laatutasot voidaan määritellä alustakohtaisesti, jolloin on mahdollista optimoida peli paremmin monelle alustalle. Kuvan 19 kohdassa Levels olevat sarakkeet määrittävät kohdealustan ja rivit halutun laatutason. Jokaisen laatutason asetukset määritetään erikseen, tosin niiden oletusarvotkin toimivat yleensä hyvin. Laatutasoja voi myös halutessaan lisätä.

Unityn alustakohtaisista asetuksista (kuva 20) tulee tarkistaa, että vähintään staattinen ja dynaaminen batching (ks. luku 4.3 Piirtokäskyt ja batching) ovat päällä. Täältä asetetaan myös julkaistaessa käytettävät ohjelmakuvakkeet ja alkunäytön kuva sekä yhtiön ja tuotteen nimi. Alusta valitaan Cursor Hotspot -kohdan alla olevista kuvakkeista.

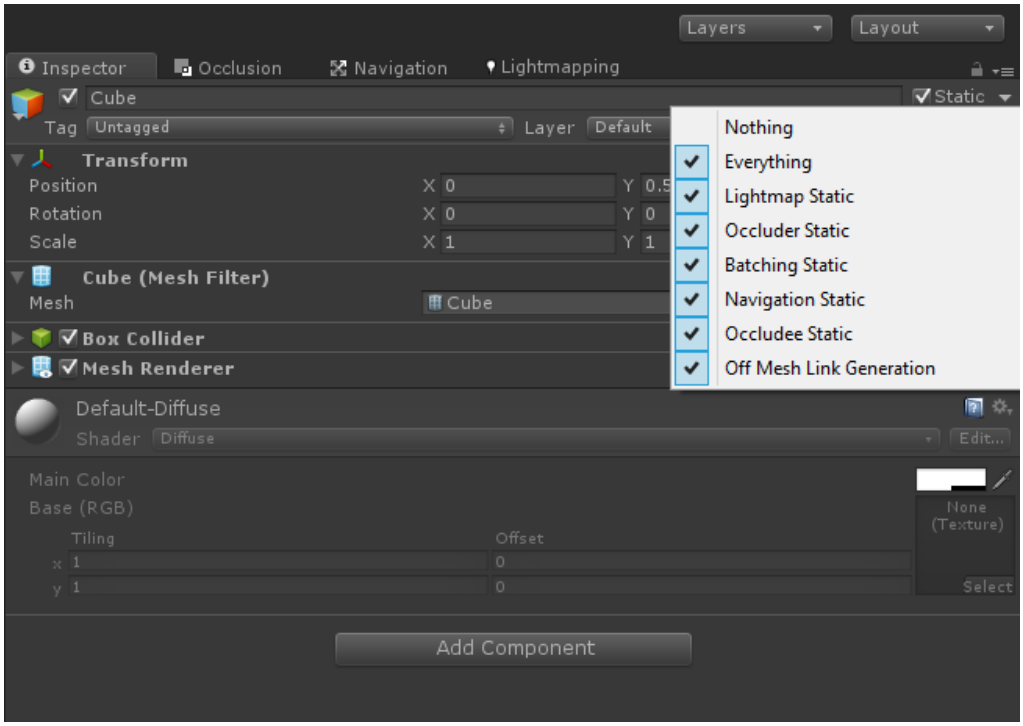


Kuva 20. Unityn alustakohtaiset asetukset.

4.3 Piirtokäskyt ja batching

Piirtokäskyjen (draw calls) määrä vaikuttaa suuresti prosessorin kuormitukseen. Jokainen esine, jonka kamera joutuu renderöimään, aiheuttaa piirtokäskyn grafiikkarajapinnalle. Suuri määrä pieniä ja yksinkertaisiakin esineitä voi alentaa kuvataajuutta huomattavasti, joten käytetään batching-tekniikoita, jotka yhdistävät esineiden geometriaa ja luovat suurempia kokonaisuuksia. [25.]

Static Batching -tekniikka toimii Unityssä automaattisesti esineille, jotka eivät liiku ja jotka on merkitty staattisiksi (Static, kuva 21). Tekniikka yhdistää peliobjektien meshit isommiksi kokonaisuuksiksi, jolloin ne pystytään renderöimään nopeammin. Nämä yhdistetyt meshit tosin säilytetään muistissa, joten joskus voi kannattaa jättää static-merkintä pois muistin säästämiseksi. Dynamic Batching puolestaan toimii Unityssä automaattisesti. Siinä pienien liikkuvien meshien verteksit lähetetään prosessorille, joka yhdistää ne. [25.]



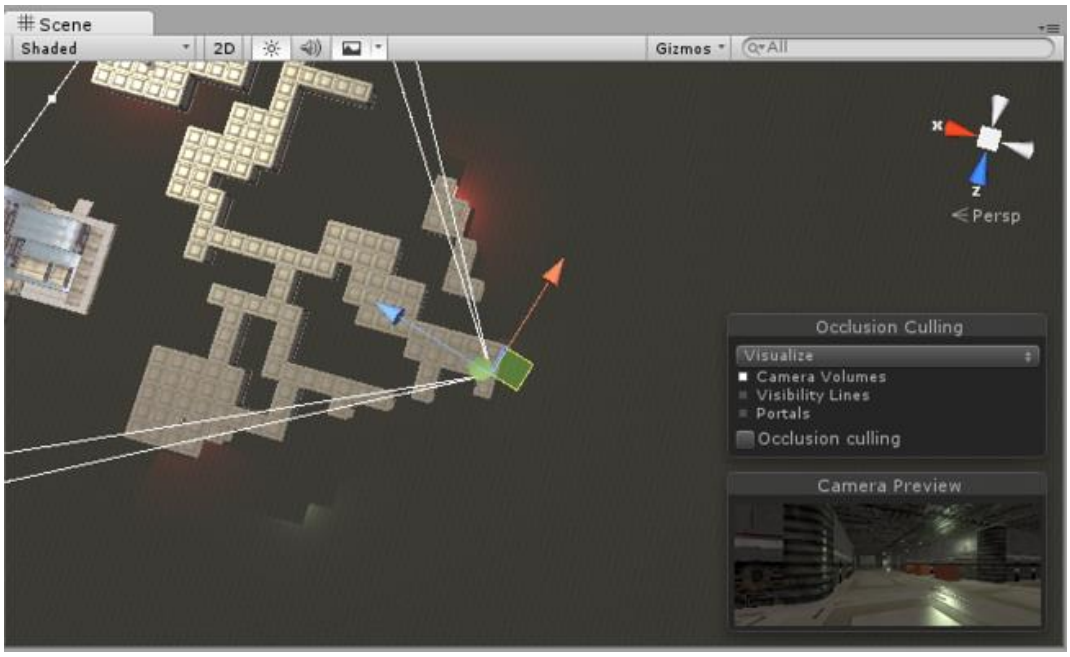
Kuva 21. Peliobjektin asetukset Unity-pelimoottorissa.

Molemmat batching-tekniikat toimivat vain peliobjekteilla, joilla on sama materiaali. Tämän vuoksi kannattaa pyrkiä käyttämään samoja materiaaleja mahdollisimman monessa peliobjektissa. Mikäli halutaan samaan materiaaliin eri tekstuureja, ne voidaan yhdistää yhdeksi suureksi tekstuurikartaksi. [25.]

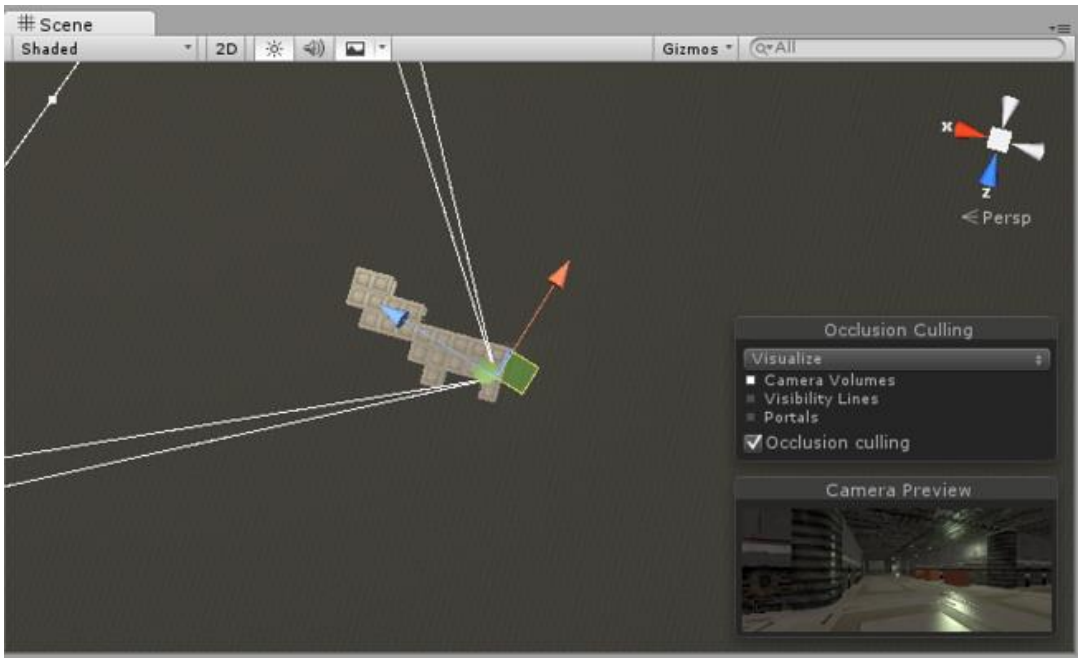
4.4 Occlusion Culling ja tarkkuustasot

Unityssä kamera renderöi kuvan taka-alalta etualalle, joten usein renderöidään peliobjekteja, joita pelaaja ei oikeasti näe. Esimerkiksi kameran osoittaessa talon seinään renderöidään myös kaikki sen taakse jäävät peliobjektit, vaikka pelaaja näkee vain seinän. Tämä vie turhaan resursseja ja voi hidastaa peliä. Occlusion culling on tekniikka, jolla määritetään piiloon jäävät peliobjektit ja jätetään ne renderöimättä.

Kuvista 22 ja 23 voidaan nähdä, kuinka kameran näkemä kuva oikeassa alareunassa pysyy täysin samana, vaikka kamera renderöi vain murto-osan peliobjekteista. Optimoidessa täytyy kuitenkin muistaa, että myös peliobjektien piilottaminen vaatii laskentaa. Olennaista onkin löytää keskitie laskentaan käytettävien ja renderöinnissä säästettävien resurssien välillä.

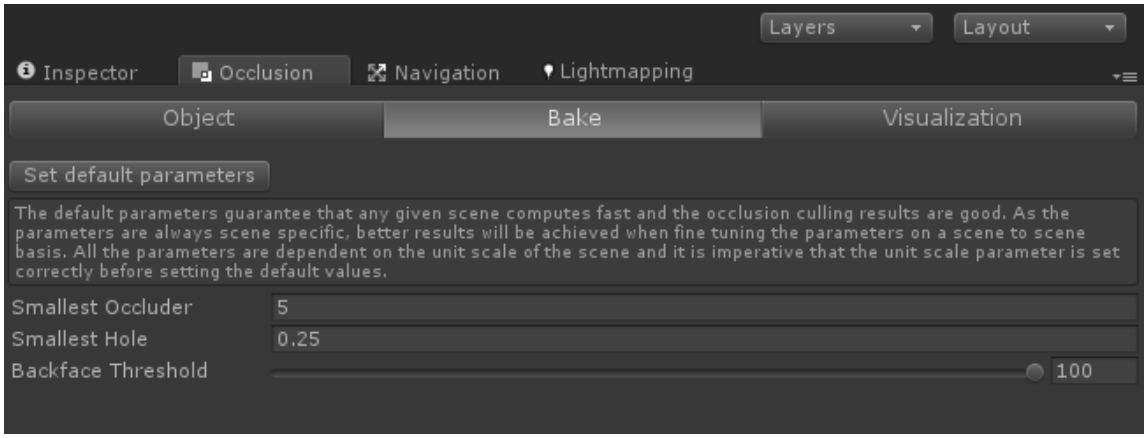


Kuva 22. Kameran piirtämät objektit ilman Occlusion cullingia [27].



Kuva 23. Kameran piirtämät objektit Occlusion Cullingin kanssa [27].

Jos peliobjekti halutaan piilottaa Occlusion Cullingilla, tulee Occludee static -asetus olla päällä. Jos taas peliobjektin pitää piilottaa muita, tulee Occluder static -asetus olla päällä. Nämä löytyvät jokaisen peliobjektin asetuksista Static-valikon alta (kuva 21). Suuret peliobjektit, kuten rakennukset, tulee asettaa piilottamaan muita, kun taas pienet peliobjektit halutaan vain piilottaa.



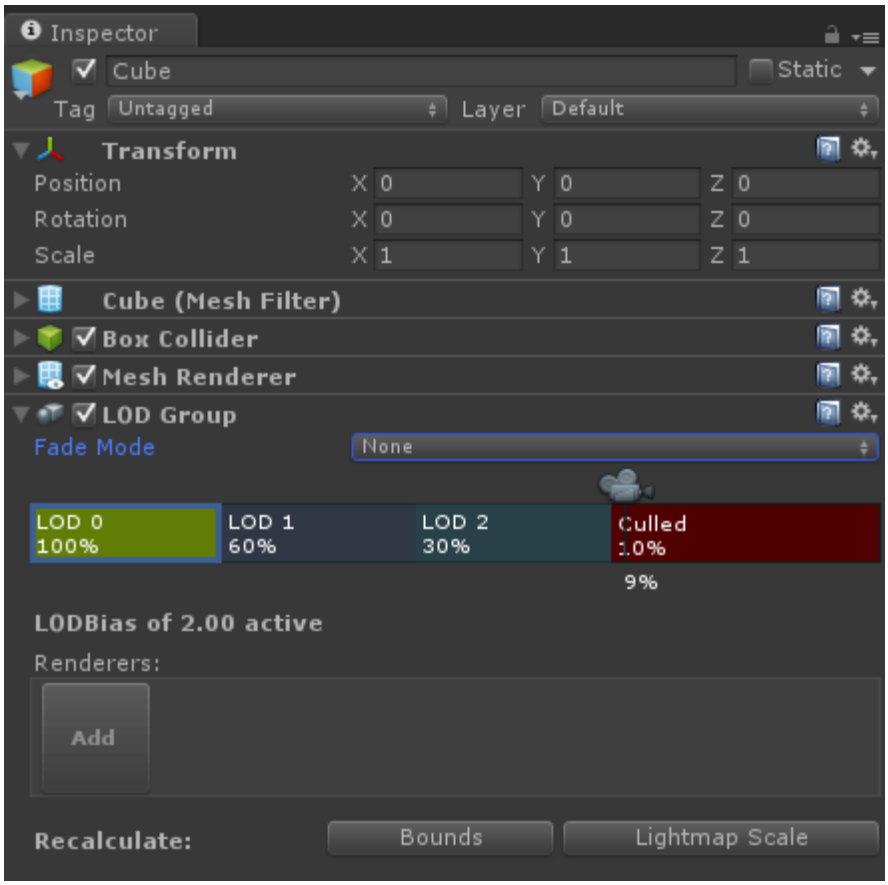
Kuva 24. Occlusion cullingin asetukset Unity-pelimoottorissa.

Itse Occlusion Cullingin laskenta tehdään Occlusion-valikosta (kuva 24). Tärkeimmät asetukset löytyvät Bake-välilehden alta ja ovat seuraavat:

- Smallest Occluder: Määrittää, minkä kokoiset peliobjektit voivat piilottaa toisia peliobjekteja. Pelitasosta riippuen arvo tulee säätää sopivaksi suhteessa peliobjektien kokoon.
- Smallest Hole: Pienin reikä, josta kamera voi nähdä läpi. Jos tasossa on peliobjekteja, joissa on reikiä (muun muassa ikkunat, aidat yms.), tulee arvo säätää suhteessa niiden kokoon. Jos arvo on liian suuri, ei kamera renderöi peliobjekteja, joiden pitäisi näkyä. Toisaalta liian pienet arvot voivat aiheuttaa huomattavasti turhaa laskentaa.

Kun nämä asetukset ovat mieluisat, tehdään varsinainen laskenta valikon alareunassa olevasta Bake-napista. Jos pelitaso on suuri, voi laskenta viedä useita minuutteja. Kun laskenta on valmis, toimii Occlusion Culling automaattisesti, jos kameran asetusten (kuva 16) kohta ”Occlusion Culling” on valittuna.

Tarkkuustasojen toteuttamiseen tarvittava komponentti on nimeltään LOD Group (kuva 25). Siinä voidaan määritellä rajat, joissa näytettävä peliobjekti vaihdetaan. Jotta tarkkuustasot voidaan toteuttaa, täytyy malleista olla valmiiksi tehtynä versiot eri tarkkuustasoille. Valmiiden mallien muokkaaminen ei onnistu Unityssä, vaan siihen vaaditaan 3D-mallinnusohjelma.



Kuva 25. Tarkkustasojen asetuset Unity-pelimoottorissa.

LODGroup-komponentin tasoihin voidaan lisätä peliobjektit (jotka sisältävät 3D-mallit) klikkaamalla ensin haluttua tasoa (esim. LOD 1) ja sitten Add-nappia Renderers-kohdan alta. Tasojen prosenttiluvut tarkoittavat pinta-alaa, jonka peliobjekti vie näytöltä. Taso Culled tarkoittaa arvoa, jolloin peliobjektia ei renderöidä lainkaan.

4.5 Koodin ja verkkoliikenteen optimointi

Projektipelissä optimoinnin pääpaino oli graafisella puolella, mutta myös koodia optimoitiin projektin edetessä. Suurin osa koodin optimoinnista tulikin sivutuotteena parempien ohjelmointitapojen omaksumisen yhteydessä. Insinöörityön ohjelmointi toteutettiin pääosin C#-ohjelmointikielellä ja käyttäen Unityn mukana tulevaa MonoDevelop-ohjelmointiympäristöä.

Suurin osa pelikoodista sijoittuu Unityssä yleensä Update-funktion sisään, joka suoritetaan jokaisessa framessa eli kuvassa, joka näytölle renderöidään. Jos kuvataajuus on

30, suoritetaan Update-funktion noin 33 ms:n välein. Tämän takia kannattaa suunnitella tarkkaan, mitkä asiat vaativat päivitystä kymmeniä kertoja sekunnissa. Yleensä esimerkiksi pelaajan syöttämät käskyt ja hahmon liikuttaminen halutaan päivittää mahdollisimman usein, jotta peli tuntuu responsiiviselta eli pelaajan toiminta näkyy näytöllä viiveettä. Sulavan liikkeen varmistamiseksi suositellaan kameroiden liikkeen suorittamista LateUpdate-funktiossa, joka suoritetaan aina Update-funktion jälkeen.

Peliobjektien etsiminen pelitasosta tai komponenttien etsiminen peliobjekteista on hidasta, joten on suositeltavaa minimoida hakukomentojen määrä. Sen sijaan viittaus peliobjektiin tai komponenttiin, jota halutaan käyttää, kannattaa tallentaa muuttujaan. Tätä muuttujaa voidaan käyttää jatkossa, koska se on nopeampaa kuin hakukomennon suorittaminen. [28.]

Peliobjektien luominen vaatii merkittävästi muistia, joten peliobjektien kierrättäminen (Object Pooling) on hyvä tapa säästää käyttömuistia. Tämä tarkoittaa, että peliobjektit (esim. luodit) luodaan kerran pelin alussa ja niitä säilytetään pelialueen ulkopuolella. Kun objekteja tarvitaan, ne siirretään haluttuun paikkaan ja sieltä takaisin säilöön, kun niitä ei enää tarvita. Näin säästetään objektien luomisen ja tuhoamisen kustannuksia. Koska objektit ovat olemassa alusta asti, ne myös varaavat muistia, joten objektien kierrättäminen kannattaa vain, jos niitä tarvitaan usein.

Insinöörityössä käytettiin verkko-ominaisuuksien toteuttamiseen Photon Unity Networking -palvelua. Siinä ensimmäinen pelaaja luo huoneen, johon muut voivat myöhemmin liittyä. Ensimmäisestä pelaajasta tulee niin sanottu masterclient, joka on vastuussa pelin yleisistä ominaisuuksista, kuten pisteidenlasku ja neutraalit viholliset. Jokainen pelaaja lähettää tiedon liikkeistään muille pelaajille, joiden lokaaleissa istunnoissa muiden pelaajien ”kopiot” liikutetaan käskyjen mukaan.

Verkkoliikenteen optimoinnissa tavoitteena on vähentää lähetettävän tiedon määrää. Esimerkiksi hahmojen ampumien luotien liikettä ei lähetetä verkossa, vaan luodit simuloidaan jokaisen pelaajan tietokoneella paikallisesti.

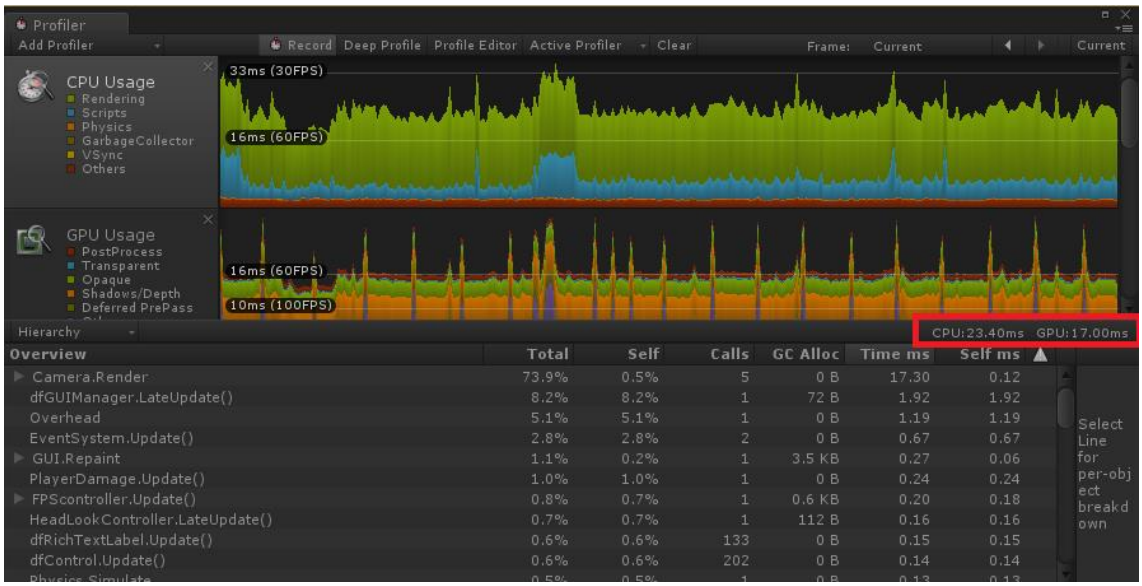
Kun pelaaja ampuu, lähetetään verkossa tieto ampumahetkestä. Tämän jälkeen luoti luodaan ja liikutetaan kaikkien pelaajien tietokoneilla paikallisesti. Mikäli alkuperäisen ampujan paikallinen luoti tulkitaan osuneeksi, lähetetään tieto osumasta uhrille (ei siis

välttämättä kaikille osapuolille). Tämä simulointitapa säästää verkkoliikenteen kuormitusta merkittävästi, mutta toisaalta menetetään hieman tarkkuutta. Jos pelaajien välillä on paljon latenssia, voi luoti osua, vaikka uhrista se näyttäisi menevän ohi.

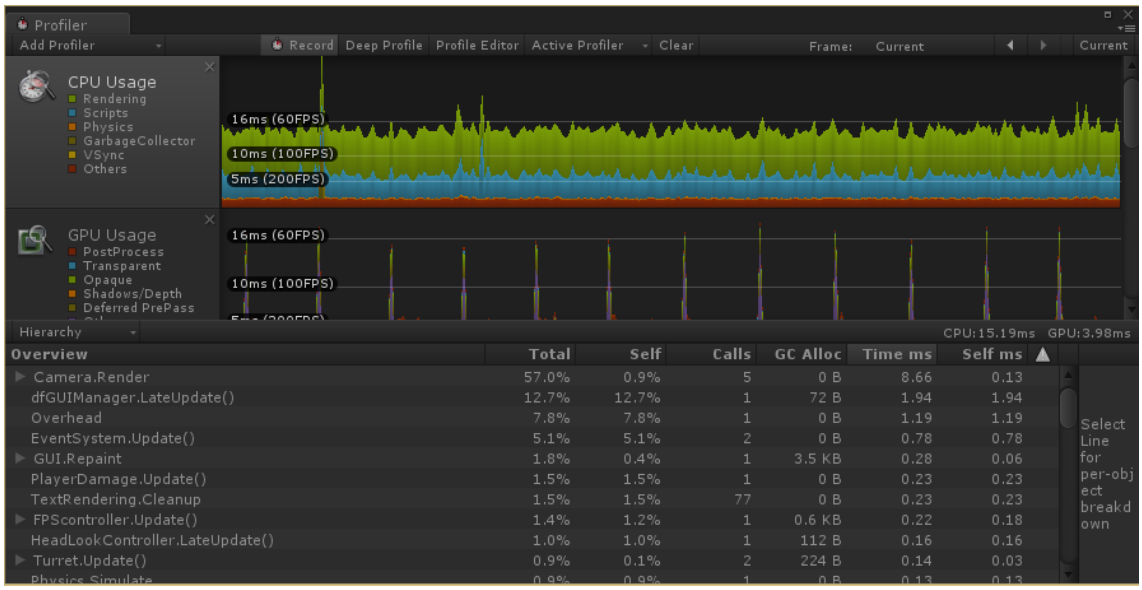
4.6 Tulokset

Optimointiprosessi aloitettiin projektipelissä toden teolla, kun huomattiin, että uuden pelitason lisääminen laski kuvataajuuden pahimmillaan noin 15 kuvaan sekunnissa. Tämä teki pelistä epämiellyttävän pelata, sillä liian alhainen kuvataajuus tekee liikkeestä nykivää.

Suurimmat yksittäiset suorituskykyyn vaikuttaneet asiat olivat Occlusion culling ja valokartat. Näiden tekniikoiden hyödyntäminen nosti kuvataajuutta merkittävästi. Koska peleissä resurssivaatimukset vaihtelevat suuresti sen mukaan, mitä kamera katsoo, on mitattavia tuloksia vaikea saada. Kuvat 26 ja 27 ovat otteita Profiler-työkalusta samasta kohtaa pelikenttää ja samaa laitteistoa käyttäen. Näin tulokset ovat mahdollisimman vertailukelpoisia. Kuvassa 26 punaisella merkitystä alueesta nähdään, että prosessorin (CPU) viemä laskenta-aika kyseisellä ajanhetkellä on 23,40 ms. Tästä kameroiden renderöinti vie suurimman osan eli 17,30 ms. Näytönohjaimen (GPU) viemä laskenta-aika puolestaan on 17,00 ms.



Kuva 26. Ote prosessorin ja näytönohjaimen suorituskyvystä ennen valokarttoja ja Occlusion Cullingia.



Kuva 27. Ote prosessorin ja näytönohjaimen suorituskyvystä muutosten jälkeen

Kun sekä valokartat että Occlusion Culling oli laskettu tasoon, saatiin selvä parannus laskenta-aikoihin. Kuvasta 27 nähdään, että uudet ajat ovat 15,19 ms (CPU) ja 3,98 ms (GPU). Tämä tarkoittaa, että prosessorin yhden framen laskemiseen tarvitsema aika laski noin 35 % ja näytönohjaimen tarvitsema aika jopa noin 75 %. Myös piirtokäskyjen määrä parannusten myötä laski noin 60 % kyseisessä kohdassa pelialuetta.

On kuitenkin otettava huomioon, että nämä arvot ovat vain esimerkkejä yhdestä ajanhetkestä ja kohdasta pelimaailmassa. Mittauspaikka pelimaailmassa oli tietoisesti valittu olemaan graafisesti raskas, ja vähemmän suorituskykyä vaativissa osissa pelimaailmaa prosentuaaliset tulokset olisivat todennäköisesti olleet pienemmät. Tarkkojen arvojen saaminen suorituskyvyn paranemisesta vaatisi koko pelialueen suorituskyvyn mittaamista pitkän ajanjakson ajalta ja saadun informaation vertailemista. Kuvat 26 ja 27 antavat kuitenkin riittävän tiedon siitä, että optimointitoimenpiteet toivat merkittävän parannuksen suorituskykyyn tietyissä olosuhteissa. Optimoinnin vaikutus oli selvästi huomattavissa myös peliä pelattaessa, sillä liikkeet olivat sulavampia paremman kuvataajuuden ansiosta.

Muut optimointitoimenpiteet, kuten koodin muokkaus, toivat myös hyötyjä. Se ei tosin näkynyt kovinkaan paljoa keskimääräisessä kuvataajuudessa. Sen sijaan hyöty näkyi suorituskyvyn kasvamisena erikoistilanteissa, kuten monen pelaajan ampuessa. Tällä alueella tosin pelissä riittää vielä parannettavaa, ja esimerkiksi ammusten kierrättäminen on yksi tavoitteista, joka jäi toistaiseksi saavuttamatta. Sillä parannettaisiin ennestään

suorituskykyä ammuntatilanteissa, joissa joudutaan luomaan ja tuhoamaan useita luoteja lyhyessä ajassa.

Monet projektipelin malleista olisivat myös voineet olla paremmin optimoituja, mutta koska suurin osa malleista ostettiin ulkopuolisilta kehittäjiltä, ei tähän ollut vaikutusmahdollisuutta. Samasta syystä projektipelissä käytettiin suurta määrää tekstuureita, ja paremmalla suunnittelulla niiden määrää olisi voitu pienentää ja näin parantaa suorituskykyä.

5 Yhteenveto

Projektipelissä prosessorin laskenta-aikaa saatiin vähennettyä noin 35 % ja näytönohjaimen jopa 75 %. Suurimmat suorituskykyyn vaikuttaneet toimenpiteet olivat valokarttojen luonti ja Occlusion Cullingin laskeminen. Projektipelin suorituskykyä olisi voinut edelleen parantaa muun muassa optimoimalla pelissä käytettyjä 3D-malleja ja toteuttamalla ammuksille peliobjektien kierrättämisen.

Pelien suorituskyvyn optimoinnissa on kyse suurten suorituskyvyn pullonkaulojen löytämisestä ja ratkaisemisesta. Sitä tarvitaan, koska nykypäivän pelien oletetaan olevan näyttäviä ja toimivan sujuvasti. Pelikokemus kärsii huonosta optimoinnista, jos esimerkiksi kuvataajuus laskee liian alas. Optimoinnissa täytyy usein löytää kompromissi visuaalisen näyttävyyden ja tavoitekuvataajuuden välillä, jotta saadaan yhtenäinen peli.

Kohdealustasta riippuen pelikehityksessä asetetaan minimivaatimukset pelin tarvitsemiin resursseihin. Optimointia tarvitaan, jotta saadaan peli toimimaan ja näyttämään hyvältä myös pienillä resursseilla. Onnistuneella optimoinnilla voidaan tietokonepeleissä laajentaa mahdollista loppukäyttäjien määrää. Konsoli- ja mobiilipelejä optimoitaessa tavoitteena on hyödyntää laitteen suorituskyky mahdollisimman hyvin.

Parhaat tulokset saadaan, kun optimointi muistetaan jokaisessa kehitysvaiheessa. Esimerkiksi 3D-mallit on hyvä optimoida jo ennen pelimoottorille tuomista ja tehdä niille tarvittavat tarkkuustasot. Materiaalien jakamisella ja tekstuurien yhdistämisellä voidaan pie-

nentää tarvittavien piirtokäskyjen määrää ja säästää näytönohjaimen rasituksessa. Muistia ja tallennustilaa voidaan säästää muun muassa pakkaamalla kuva- ja äänitiedostot oikein.

Pelimoottorien työkaluja hyödyntämällä voidaan tunnistaa ongelmakohdat ja parantaa pelin suorituskykyä huomattavasti. Käytettävissä oleviin työkaluihin tutustuminen ja kohdealustojen tarkka määrittely projektin alkuvaiheessa helpottaa optimointia huomattavasti.

Lähteet

1 Video Game History Timeline. 2016. Verkkodokumentti. The Strong. <http://www.museumofplay.org/icheg-game-history/timeline/>. Luettu 25.3.2016.

2 A Brief History of Graphics. 2014. Verkkodokumentti. Top Documentary Films. <http://topdocumentaryfilms.com/brief-history-graphics/>. Luettu 15.3.2016.

3 Overmars, Mark. 2012. A Brief History of Computer Games. Verkkodokumentti. <http://www.cs.uu.nl/docs/vakken/b2go/literature/history_of_games.pdf>. Luettu 25.3.2016.

4 Bell, John. 2005. Vector vs. Raster Displays. Verkkodokumentti. <http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/VectorVsRaster.html>. Luettu 25.3.2016.

5 Asteroids. 2016. Verkkodokumentti. The International Arcade Museum. <http://www.arcade-museum.com/game_detail.php?game_id=6939>. Luettu 25.3.2016.

6 Zaxxon. 2016. Verkkodokumentti. The International Arcade Museum. <http://www.arcade-museum.com/game_detail.php?game_id=12757>. Luettu 25.3.2016.

7 Diskin, Patrick. 2004. Nintendo Entertainment System Documentation. Verkkodokumentti. <http://nesdev.com/NESDoc.pdf>. Luettu 25.3.2016.

8 Ki, Saikyo. 2001. Super Nintendo. Verkkodokumentti. <http://www.gamefaqs.com/snes/916396-super-nintendo/reviews/19457>. Luettu 25.3.2016

9 Quake. 2015. Verkkodokumentti. QuakeWorld. <http://wiki.quakeworld.nu/Quake>. Luettu 25.3.2016.

10 Garney, Ben & Preisz, Eric. 2010. Video Game Optimization. Boston, MA: Course Technology.

11 Crysis. 2016. Verkkodokumentti. Wikimedia Foundation <https://en.wikipedia.org/wiki/Crysis>. Luettu 25.3.2016

12 Schuller, Daniel. 2011. Game Programming for Serious Game Creation. Boston, MA: Cengage Learning.

13 Murray, Jeff W. 2013. Game Development for iOS with Unity3D. Boca Raton, Florida: CRC Press.

14 Hu, Peng & Zhu, Kai. 2014. Strategy research on the performance optimization of 3D mobile game development based on Unity. Journal of Chemical and Pharmaceutical research 6/2014, s. 785–791.

15 Silverman, David. 2013. 3D Primer for Game Developers: An Overview of 3D Modeling in Games. Verkkodokumentti. <http://gamedevelopment.tutsplus.com/articles/3d-primer-for-game-developers-an-overview-of-3d-modeling-in-games--gamedev-5704>. Luettu 25.3.2016.

16 Optimizing Graphics Performance. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>. Luettu 25.3.2016.

17 Bump and Normal Maps. 2016. Verkkodokumentti. Blender Foundation. <http://wiki.blender.org/index.php/Doc:2.4/Manual/Textures/Influence/Material/Bump_and_Normal>. Luettu 25.3.2016

18 LOD Group. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/class-LODGroup.html>. Luettu 25.3.2016.

19 Lightmapping In-Depth. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/460/Documentation/Manual/LightmappingInDepth.html>. Luettu 25.3.2016.

20 West, Mick. 2006. Mature Optimization. Game Developer Magazine 1/2006.

21 Gribble, Paul. 2012. C Programming Bootcamp. Verkkodokumentti. <http://www.gribblelab.org/CBootCamp/index.html>. Luettu 25.3.2016.

22 Understanding Automatic Memory Managements. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html>. Luettu 25.3.2016.

23 Vestola, Mikko. 2007. Image compression. Verkkodokumentti. Teknillinen Korkeakoulu. <http://www.mvnet.fi/index.php?osio=Tutkielmat&luokka=Yliopisto&sivu=Image_compression>. Luettu 25.3.2016.

24 Light Troubleshooting and Performance. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/LightPerformance.html>. Luettu 25.3.2016.

25 Global Illumination. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/GIIntro.html>. Luettu 25.3.2016.

26 Draw Call Batching. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/DrawCallBatching.html>. Luettu 25.3.2016.

27 Occlusion Culling. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/Manual/OcclusionCulling.html>. Luettu 25.3.2016.

28 Overview: Performance Optimization. 2016. Verkkodokumentti. Unity Technologies. <http://docs.unity3d.com/410/Documentation/ScriptReference/index.Performance_Optimization.html>. Luettu 25.3.2016.