Antti Veräjänkorva

Mobiilipelin erikoistehosteet

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

4.3.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Antti Veräjänkorva Mobiilipelin erikoistehosteet 55 sivua + 2 liitettä 4.3.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Lehtori Antti Laiho, yliopettaja Harri Airaksinen

Insinöörityön tarkoituksena oli luoda erikoistehosteet valmisteilla olevaan mobiilipeliin. Tavoitteena oli luoda tarvittavat tehosteet pelin ensimmäiseen pelattavaan esittelyversioon. Lopullisen pelin tehosteet voivat siis erota merkittävästikin tässä työssä esitellyistä tehosteista.

Reaaliaikaisissa peleissä on alusta alkaen ollut tehosteita joiden tarkoitus on ollut viestiä pelaajalle pelin tapahtumia ja antaa pelaajalle palautetta hänen tekemistään toiminnoista. Tehosteiden näyttävyys on kehittynyt tietokoneiden nopeuden kehittyessä ja pelien yleisen kehityksen myötä. Tehosteiden näyttävyys ja tyyli muokkaantuu myös aikakauden tyylin ja alustan mukana.

Tehosteiden luomisessa kiinnitettiin erityistä huomiota siihen, että kyseessä on mobiilipeli, jolloin näytön koko ja pelitilanne voi olla hyvin erilainen kuin jos peliä pelattaisiin olohuoneessa. Tehosteiden luonnissa ajateltiin paitsi niiden näyttävyyttä, niiden funktionaalista tarkoitusta ja sitä, mitä niillä on tarkoitus pelaajalle viestiä.

Erityistä huomiota työssä kiinnitettiin myös tehosteiden visuaalisuuden takana olevaan matematiikkaan ja algoritmeihin. Työssä luotiin paitsi itse visuaalinen grafiikka, myös toimintalogiikka.

Työssä ei otettu huomioon tehosteiden varsinaista käyttöä pelissä ja siihen liittyviä teknisiä ratkaisuja ja rajoituksia.

Insinöörityön tuloksena valmistuivat mobiilipelin erikoistehosteet, ja ne liitettiin pelin esittelyversioon. Tehosteet olivat muun muassa vahinkotehoste, aseen leimahdus, parannustehoste, laavavirtaus, valojuovat ja monia muita. Kaikki tehosteet liitettiin peliin, sen sisäisen tehostejärjestelmän kautta.

Avainsanat	tehoste, mobiilipeli, Unity

Author Title Number of Pages Date	Antti Veräjänkorva Special effects in a mobile game 55 pages + 2 appendices 4 March 2015
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Antti Laiho, Senior Lecturer Harri Airaksinen, Principal Lecturer

The purpose of the project described in this thesis was to create special effects for a mobile game which is currently in development. The goal was to create all necessary effects for game's first playable demonstration. The effects in the final game may be vastly different than the ones which are presented in this thesis.

In real-time games have had effects from very beginning, which were used to communicate game's events and give feedback to the player's actions. Faster computers and general evolution of games have improved visual fidelity of the effects. Visuals are also been effected by the style of current era and platform where effects have been used.

When creating these effects in this project, special attention was given to the fact that the game is a mobile game, where screen size and gaming situation can be very different, than if player was playing in his living room. Attention was also given to functional purpose of the effects and what kind of message the effects should communicate to the player.

This thesis studies in detail the mathematics and the algorithms behind the effects. This project created the visual component for the effects, but also program logic behind it. The thesis does not take into account how effects are actually used in the game or technical details and restrictions related to it.

As end result of this project, special effects were created and implemented into the mobile game's demonstration version. The effects included but not limited to damage effect, weapon muzzle, healing effect, lava flow and tracer effect. All the effects were implemented via the game's internal effect system.

Keywords	effect, mobile game, Unity

Sisällys

Lyhenteet

CPU	Centeral Processing Unit eli keskusprosessori. Tietokoneen prosessointiyksikkö, joka on vastuussa tietokoneen pääasiallisesta laskennasta.
GPU	Graphics Processing Unit eli grafiikkaprosessori. Tietokoneen grafiikkaprosessointiyksikkö, joka on vastuussa grafiikan piirrosta.
C#	C Sharp -ohjelmointikieli. Microsoftin kehittämä ohjelmointikieli, jota käytetään muun muassa Unity-pelimoottorissa.
CG	Lyhenne tulee sanoista C for Graphics. Kieli on Nvidian kehittämä, C-kielen syntaksiin pohjautuva varjostimille tarkoitettu ohjelmointikieli.
HLSL	High-Level Shader Language. Microsoftin kehittämä varjostinohjelmointikieli. Kieli on kehitetty Mircosoftin Direct3D-grafiikkarajapintaa varten.
GLSL	OpenGL Shading Language. Kielen on kehittänyt OpenGL Architecture Review Board. Kieli on tarkoitettu OpenGL-grafiikkarajapintaa varten.
UV	Pisteen koordinaatti tekstuuriavaruudessa.
DLC	Downloadable Content eli peliin ladattava lisäosa.
API	Application Programming Interface eli ohjelmointirajapinta. Ohjelmointirajapinnan avulla voidaan luoda ohjelmistoja, jotka hyödyntävät jo valmiiksi tehtyä ohjelmointirajapintaa.

1 Johdanto

Elokuvissa erikoistehosteet ovat arkipäivää. Liki jokainen elokuva sisältää jonkinlaisia erikoistehosteita, joko fyysisiä tai digitaalisia. Myös pelit sisältävät erikoistehosteita, räjähdyksiä, lasersäteitä, laavaa, tulta, roiskeita, mitä kulloinenkin peli tarvitsee luodakseen visuaalista informaatiota, haluttua ilmettä ja tunnelmaa.

Tämän insinöörityön tarkoituksena on tutkia, millaisia tehosteet olivat ennen ja millaisia ne ovat nyt. Tarkoitus on tutkia laajalti ja eri alustoilta ja erityyppisistä peleistä, millaisia vastaavia tehosteita niissä on käytetty selvittää millaisia tekniikoita ja algoritmeja tehosteissa käytetään ja lopuksi luoda erikoistehosteet valmisteilla olevaan Remedy Entertainmentin mobiilipeliin.

Remedy Entertainment, kuvassa 1, on suomalainen pelikehitysyritys, joka on perustettu vuonna 1995. Tunnetuimpia tuotteita ovat Max Payne ja Alan Wake. Yrityksen ensimmäinen peli oli Apogeen julkaisema Death Rally, josta myöhemmin tehtiin myös mobiiliversio Death Rally HD. (Remedy Games, 2014.)

Kuva 1. Remedy Entertainmentin aula (Remedy Games, 2014).

Remedy Entertainment työllistää noin 150 henkeä, ja yrityksen liikevaihto vuonna 2013 oli noin 10,5 miljoonaa euroa. Nyt Remedy kehittää Xbox One -peliä Quantum Break ja vielä salaista iOS-peliä, projektinimellä P2, johon tässä insinöörityössä tehdään tehosteita. (Remedy Games, 2014.)

P2 on valmisteilla oleva, Unitylla tehty iOS-peli, jossa pelaaja matkaa läpi ”pimeän” Amerikan. Päähahmo on pelissä kuollut ja joutunut jonkinlaiseen kiirastuleen, josta pois päästäkseen on taivallettava pitkä matka läpi ”pimeän” Amerikan, kerättävä resursseja, taisteltava ja haalittava mukaansa lisää sankareita, jotka auttavat päähahmoa taistelussa. Päähahmoja on kaksi, mies ja nainen, joiden väliltä pelaaja voi valita, kummalla peliä pelaa. Mukana oleva joukko muita sankareita ovat samoja molemmilla päähahmoilla, joskin niiden tarinallinen sisältö saattaa hieman vaihdella. Taistelu käydään kolmen sankarin tiiminiä, jotka yhdessä käyvät taisteluun demoneja ja hirviöitä vastaan (kuva 2). Eri taistelutoimintoja tehdään raahaamalla toimintoa kuvaava ikoni vihollisen päälle. (Pirinen 2015.)

Kuva 2. Konseptikuva pelin taistelusta.

Taistelu on vuoroittaista. Ensin pelaaja aloittaa ja tekee siirtonsa annetussa ajassa. Tämän jälkeen vihollinen tekee siirtonsa tekoälyn avulla. Jos pelaaja voittaa, hän pääsee etenemään matkallaan pimeän Amerikan läpi. (Pirinen 2015.)

2 Tietokonepelien tehosteiden historia

Tietokonepeleissä on kautta aikain ollut tehosteita. Ensimmäisiä reaaliaikaisia tietokonepelejä oli Spacewar!, Massachusetts Institute of Technologyssä kehitetty peli, jonka kehitti Steve Russell vuonna 1962 PDP-1-tietokoneelle (kuva 3). (Spacewar! 2005.)

Kuva 3. Spacewar!-peli (Spacewar! 2005).

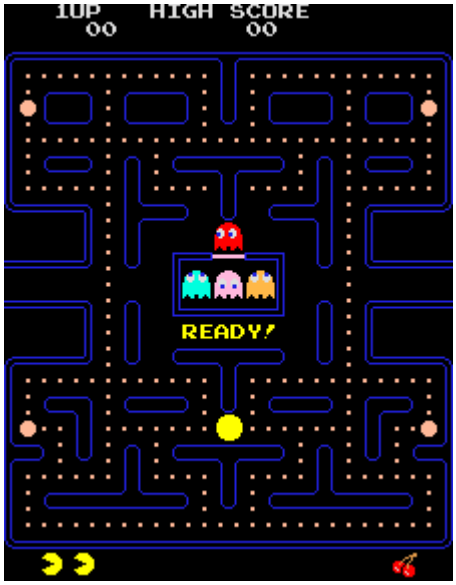
Spacewar!-pelissä on kaksi avaruusalusta, jotka yrittävät tuhota toisensa ja jotka samalla välttelevät joutumasta ruudun keskellä olevaan mustaan aukkoon. Grafiikka on viivoja ja pisteitä, mutta aluksen tuhouduttua se esimerkiksi räjähtää pisteiksi, mikä on selvästi tehoste. (Spacewar! 2005.)

Kymmenen vuotta Spacewar!-pelin jälkeen ilmestyi Magnavox Odyssey, kuvassa 4. Se oli ensimmäinen kotiin ostettavissa oleva pelikonsoli, ja sen kehittivät Ralph Baer, Bill Harrison ja Bill Rush vuonna 1972. (Winter 2013.)

Kuva 4. Odyssey-pelikonsoli (Winter, 2013).

Laite pystyi piirtämään vain yhtä väriä ja sitäkin vain muutamia pikseleitä. Niinpä suurin osa grafiikasta tulee televisioruudun päälle asetettavasta kalvosta, kuten nähdään kuvasta 4. Kalvot toimivat pelin grafiikkana, ja laitteen piirtämät muutamat pisteet näkyivät kalvon läpi. Nämä kalvot olivat tavallaan fyysisiä tehosteita. (Winter 2013.)

Vuonna 1980, eli kahdeksan vuotta Odysseyn aikojen jälkeen, Namco kehitti Pac-Man-pelin pelihalleihin, kuvassa 5. Pac-man on sittemmin kopioitu useille eri alustoille. (Pac-Man 2014.)



Kuva 5. Alkuperäinen Pac-man-pelihallipeli (Pac-Man 2014).

Pac-Man on peli, jossa pieni keltainen pallo syö keltaisia pisteitä. Pelissä on jo paljonkin eri tehosteita. Hahmoilla on jopa animaatiota, joskin hyvin yksinkertaisia, mutta kuitenkin hahmoissa on liikettä. Ne myös vaihtuvat erinäköisiksi, jos pelaaja syö oikeanlaisen esineen. Pelaajan suoritettua koko tason alkavat tason seinät välkkyä. Kaikki nämä ovat tehosteita, ja osa niistä on edelleen toimivia. (Pac-Man 2014.)

Kaksitoista vuotta Pac-manin ilmestymisen jälkeen, vuonna 1992, Id Software julkaisi Wolfenstein 3D-pelinsä, kuvassa 6. Peli piirsi uskottavaa 3D-grafiikkaa, vaikka se oli oikeastaan vain puoliksi 3D, niin sanottu 2.5D tai pseudo-3D, mikä tarkoittaa sitä, että peli on kolmessa ulottuvuudessa, mutta liikkuminen on rajoitettu 2D-tasolle. (Hastings 2014.)

Kuva 6. Wolfenstein 3D -peli (Hastings 2014).

Tehosteet olivat vain kuvia, kuten myös esineet ja hahmot. Hahmojen animaatiot olivat Pac-Manin tapaan muutaman kuvan kuvasarjoja. Ruutu välähteli osumista, ja oman hahmon kuvake myös kuvasti hahmon terveydentilaa. Pelissä oli jopa valotehosteita, ja kun tehtävä oli suoritettu, ruutu muuttui tasaisesti himmentyen mustaksi. (Hastings 2014.)

Noin kymmenen vuotta Wolfenstein 3D:n ilmestymisen jälkeen alkoivat grafiikkakiihdyttimet yleistyä kotitietokoneissa, ja ne nopeuttivat merkittävästi pelien grafiikan kehitystä. Remedy Entertainmentin kehittämä Max Payne, kuvassa 7, julkaistiin vuonna 2001. Peli hyödynsi grafiikkakiihdytystä, joka mahdollisti yksityiskohtaisemman grafiikan ja monimutkaisemmat tehosteet. Max Paynessa oli tehosteita kuvasarja-animaatioista hiukkassimulaatioon. (Max Payne 2002.)

Kuva 7. Max Paynen aseen savutehoste (Max Payne 2002).

Kaikki Max Paynen tehostetekniikat ovat edelleen laajalti käytössä. Oikeastaan erona nykypeleihin on, että kaikkea oli vain vähemmän. (Max Payne 2002.)

Nopeasti grafiikkakiihdyttimen yleistymisen jälkeen niihin alkoi ilmestyä ohjelmoitavia varjostimia, joiden avulla pelintekijät voivat luoda omat tapansa piirtää pikseleitä eikä tarvitse piirtää grafiikkaprosessorin ehdoilla. Kandalaisen Bio Waren vuonna 2010, eli kymmenen vuotta Max Paynen jälkeen, julkaistu Mass Effect 2 hyödyntää laajasti ohjelmoitavia varjostimia. Peli käyttää muunneltua Unreal 3 -moottoria. (Mass Effect 2 2014.)

Kuva 8. Mass Effect 2 -peli (Mass Effect 2 2014).

Kuvassa 8 näkyy Mass Effect 2 -pelin monia tehosteita, kuten Bokeh-syväterävyys, ihon alle sirostuvaa valoa, varjoja ja hehkua. Unreal 3 -moottori tarjoaakin runsaasti grafiikkaominaisuuksia, joita tässä on listattu vain murto-osa. (Mass Effect 2 2014.)

Mobiililaitteissakin on nykyisin grafiikkakiihdytin ja teholtaan, verrattuna tietokoneisiin, ne ovat noin Max Payne -pelin aikakautta. Ne nopeutuvat koko ajan ja tuovat kaiken aikaa mobiilipelien grafiikkaominaisuuksia lähemmäs tietokoneiden ja konsoleiden ominaisuuksia.

3 Referenssitehosteet

3.1 Tehosteen luomisen vaiheet

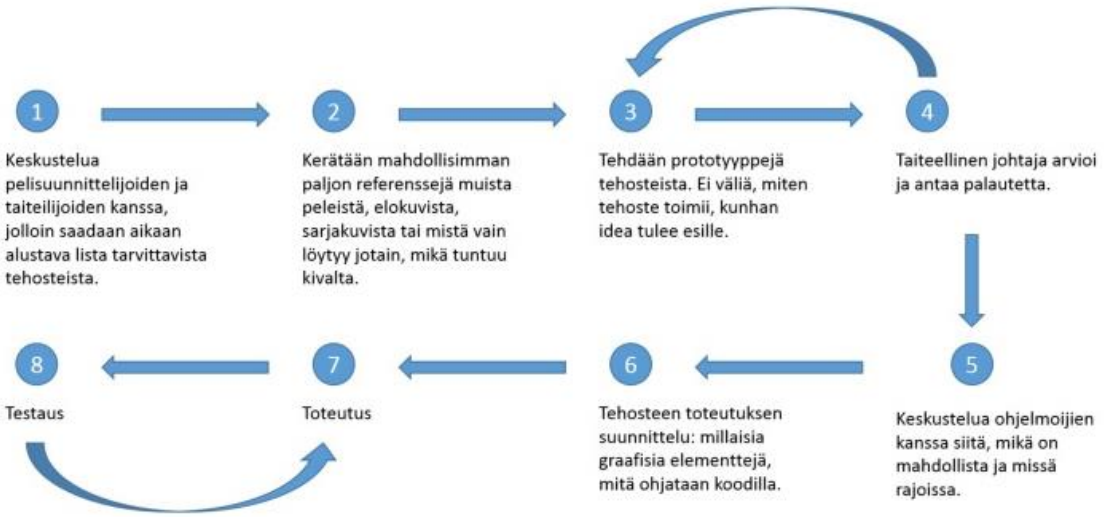
Elokuvassa tehoste on helppo selittää: se on jotain, mikä on luotu oikean kuvan päälle jollain tekniikalla. Pelissä oikeaa kuvaa ei ole, joten tehosteen määritteleminen on hankalampaa. Tavallisesti sillä tarkoitetaan räjähdyksiä, leimahduksia ja samoja visuaalisia elementtejä, jotka elokuvassakin olisivat tehosteita. Usein peleissä jopa puhutaan VFX-

tehosteista, vaikka sillä ajatuksella koko peli on yhtä isoa tehostetta. Teknisesti tehosteesta tulee tehoste, kun sitä hallitaan jonkinlaisen tehostejärjestelmän kautta. Se ei ole pelaajalle näkyvää, mutta vaikuttaa siihen, miten tehoste luodaan ja miten sitä käytetään. Niinpä pelissä saattaa olla elementtejä, jotka näyttävät tehosteelta, mutta eivät teknisesti ole tehosteita. Pelit tekevät itse omat määritelmänsä eri asioille, ja tavat ovat hyvin erilaisia. Esimerkiksi Alan Wake -pelissä hahmo on jotain, joka on määritelty hahmolistaan ja jolla on vähintään yksi luu. Se ei kuitenkaan tarkoita, että hahmon pitäisi näyttää ihmiseltä tai eläimeltä. Pelissä oli esimerkiksi kirjoituskoneen paperi, joka oli tehty hahmoksi. Pelaaja tuskin kuitenkaan mielsi paperia hahmoksi, vaikka peli teknisesti käsitteli paperin samalla tavalla kuin muutkin hahmot.

Tehosteita tehtäessä ensimmäinen asia on selvittää, millaisia tehosteita ylipäätään tarvitaan ja mihin tarkoitukseen ne ovat tulossa. Joskus on tarpeen saada tehtyä jotain mikä todistaa, että jokin on mahdollista, mutta sen ei tarvitse olla vielä valmis. Tämänkaltaisesta työstä käytetään termiä POC-vaihe (engl. Proof Of Concept). Usein pelin sisäiset esittelyversiot, eli demot, ovat juuri tämäntyyppisiä projekteja.

Pelin demot ovat joko sisäisiä tai julkisia. Sisäiset demot ovat usein versioita, joiden perusteella päätetään, annetaanko pelille lisää rahoitusta vai lopetetaanko pelin kehitys. Julkiset demot taas ovat yleisön mielenkiinnon herättämistä varten tehtyjä. Tässä työssä kyseessä on ensimmäinen sisäinen demo eli niin sanottu First Playable Demo, jonka perusteella pelin jatkokehitykseen annetaan rahaa.

Tehosteen työvaiheet ovat kuitenkin samat riippumatta siitä, ollaanko tekemässä sisäistä- vai julkista demoa, mutta julkisessa demossa annetaan paljon enemmän arvoa visuaalisuudelle, kun taas sisäisessä arvostetaan enemmän pelillistä toimivuutta. Sisäisessä demossa peliä tarkastelevat pelinkehittäjät, jotka ovat nähneet useita demoja ja tietävät, että taloa esittävä harmaa kuutio on joskus myöhemmin talon näköinenkin. Kuvasta 9 näkyy tehosteen luomisen tavanomainen työnkulku.



Kuva 9. Tehosteen luomisen työnkulku.

Käymällä suunnittelijoiden ja artistien kanssa keskusteluja pelin tehostetarpeista saa aikaan listan, jonka perusteella voidaan alkaa kerätä referenssejä muista tuotteista. Usein tehosteen varsinainen tekeminen on helpompaa kuin keksiä, millaisen tehosteen tekisi. Siksi on tärkeää kerätä paljon referenssimateriaalia tehosteita suunniteltaessa. Referenssimateriaalin perusteella voi alkaa kehittää tehosteen prototyyppiä. Se tavallisesti tehdään melko itsenäisesti, joskin taiteellinen johtaja antaa palautetta ja ohjeistaa sopivaan suuntaan.

Usein graafinen ilme ei kuitenkaan ole vielä tiedossa, kun ensimmäisiä tehosteita tehdään. Se johtaa siihen, että tehosteet tehdään jossain vaiheessa uudestaan, kun graafinen ilme alkaa löytyä. Kaikkea ei kuitenkaan tarvitse heittää pois. Usein kehitettyjä tekniikoita voidaan edelleen hyödyntää, ja muutokset tehdään usein graafisiin elementteihin. Prototyypin jälkeen keskustellaan ohjelmoijien kanssa siitä, miten paljon saa käyttää tekstuurimuistia, paljonko saa käyttää polygoneja ja partikkeleita, millainen on tehostejärjestelmä, millaiset säännöt ja mahdolliset rajoitukset ovat.

Usein tehostejärjestelmää ei ensimmäisten tehosteiden aikana ole, vaan sitä aletaan tässä kohtaa kehittää. Kun tiedetään, mitä vaatimuksia teknologialla on, voidaan suunnitella, mitä osia tehdään grafiikalla ja mitä koodissa, eli voiko peli laskea koodissa tulen liekit höyrysimulaatiolla vai tehdäänkö liekit grafiikalla tai laitetaanko liekkitekstuurit partikkeleihin yksittäisinä kuvina vai animaationa vai käytetäänkö partikkeleja lainkaan. Kysymyksiä on monia tehosteesta riippuen, ja kun niihin on löytynyt jokin ratkaisu, voidaan

tehoste toteuttaa. Testausta tehdään kaiken aikaa tehostetta luotaessa, mutta lopuksi on tärkeää tarkistaa, että tehoste toimii oikein oikeassa pelissä ja oikealla alustalla.

3.2 Iskun osuma

Pelissä on mahdollista joutua ammutuksi, lyödyksi, purruksi, raadelluksi, ja niin edelleen. Hahmoille voidaan tuottaa vahinkoa monin eri tavoin, ja aina kun vahinkoa tuotetaan, pitää jotenkin indikoida pelaajalle, että hän tuotti vahinkoa tai vastaanotti vahinkoa. Pelit tekevät tämän indikaation hyvin erilaisin tavoin, ja on tärkeää pitää mielessä pelin kohdeikäraja. Jos peli on K18, verta voidaan huoletta käyttää, mutta jos peli on K16, puhumattakaan siitä alemmasta, pitää olla tarkkana, miten paljon ja millä tavalla verta, tai muuta sellaista, näkyy. Myös alusta, jolle peliä tehdään, pitää ottaa huomioon, koska alustan omistajalla on aina sanansa sanottavana siihen, millaisia pelejä alustalla on. Eri mailla on omat säädäntönsä sille, mikä on K16 ja mikä K18. Esimerkiksi Alan Wake -pelissä ikäraja nousi Koreassa K18-luokkaan, koska yhdessä välivideossa kirveen iskun ääni oli liian vetinen ja siitä näkyi veriroiske, varsinainen isku oli rajattu pois kuvasta. Ääni muutettiin ja roiske poistettiin, jotta ikäraja pysyi alle K18:n. Kaikki muutokset kuitenkin maksavat, joten on hyvä pitää ikärajat mielessä jo aikaisessa vaiheessa. Pahimmassa tapauksessa muutos saattaisi viivästyttää pelin julkaisua.

Useat pelit näyttävät jonkinlaisen tehosteen osumakohdassa, jonkin todella nopean tähden omaisen asian. Jotkut pelit vielä väläyttävät kohdehahmoa vaaleana indikoidakseen tarkemmin, keneen osui, kuten referenssipelissä Final Fantasy 13 kuvassa 12. Toiset pelit taas tyytyvät näyttämään vain iskutehosteen ja antavat animaation osoittaa, kuka vastaanotti iskun, kuten pelit Tekken 6 kuvassa 11 tai Street Fighter 4 kuvassa 10.

Kuva 10. Capcomin Street Fighter 4 -peli (Street Fighter 4 – Official Trailer 2009).

Kuva 11. Namco Bandai Gamesin Tekken 6 -peli (Tekken 6 Gameplay 2010).

Kuva 12. Square Enixin Final Fantasy 13 -peli (Final Fantasy 13 Battle Gameplay 2010).

Ottaen huomioon mobiilinäytön koon ja sen, että fyysinen tila, jossa peliä pelataan, voi olla mitä tahansa bussista olohuoneeseen, on erityisen tärkeää, että tehoste näkyy ja näyttää sen, mitä halutaan. Hahmon indikoiminen jollain välähdyksellä on toki epärealistista eikä edes visuaalisesti mielenkiintoista, mutta se kertoo selvästi, kuka sai osuman.

3.3 Parannustehoste

Yllättävän harvassa pelissä on oikeastaan mitään varsinaista tehostetta parantamiselle. Monessa pelissä tosin voi hahmoa parantaa, ja nykytrendin mukaan hahmon tulee parantua taistelun jälkeen automaattisesti, kunhan pysyy hetken poissa taistelusta. Tälle harvemmin on kuitenkaan mitään muuta indikaatiota kuin korjaantuva kuntomittari. P2-pelissä halutaan kuitenkin antaa parantamiselle jokin visuaalinen tehoste.

Referenssipeleissä, World Of Warcraft kuvassa 13 ja Final Fantasy 7 kuvassa 14, parannusta indikoitiin hahmon ympärillä kieppuvina säteinä, mikä yksittäisen hahmon kohdalla toimiikin hyvin.

Kuva 13. World of Warcraftin parannusloistun tehoste (Corefire Imp – Heal).

Kuva 14. Final Fantasy 7 (Healing Wave).

P2-pelissä taas voi olla tilanne, jossa koko ryhmä parannetaan kerralla, jolloin samaan tapaan hahmojen ympärillä kieppuvat säteet voisivat näyttää hiukan oudolta. Niinpä tämän tehosteen kohdalla mikään referenssipeli ei tarjonnut sopivaa tyyliä.

3.4 Vauhtiviivat

P2-pelissä on tarvetta vauhtiviivoille, koska osalla hahmoista on käytössään lyömäaseita, joiden liikettä halutaan korostaa vauhtiviivalla. Varsinkin pienellä mobiilinäytöllä on tärkeää, että pelaaja näkee, mitä tapahtuu, ja lisäksi vauhtiviiva voi olla visuaalisesti mielenkiintoinen.

Hyvin tavallinen tapa, varsinkin konsolipeleissä, on tehdä fysikaalista liikkeen sumennusta, joka lisää hyvinkin realistisia vauhtiviivoja. Fysiikkaan perustuvat tavat eivät kuitenkaan anna kovin suurta taiteellista vapautta. Niinpä monesti päädytään tekemään ylimääräinen vauhtiviivatehoste, joka tehostaa lyömäaseen sivalluksen visuaalisuutta.

Referenssipelejä vauhtiviivoista oli helppo löytää, sillä pelejä, joissa tehostetta käytetään, on tarjolla paljon.

Kukin referenssipeli on visuaalisesti lähestynyt vauhtiviivaa eri tavoin. Heavenly Sword -pelin, kuvassa 15, vauhtiviivat ovat hyvinkin värikkäitä ja kauniita. Niiden tarkoitus on

enemminkin luoda kaunista taidetta kuin tehostaa sivalluksen nopeutta tai voimaa. Ninja Gaiden, kuvassa 16, taas pyrkii tehostamaan liikettä. Pelissä miekan sivallus on todella nopea, ja hädin tuskin sitä ehtisi edes huomata ilman vauhtiviivatehostetta. Zeldassa, kuvassa 17, vauhtiviiva on visuaalinen ja funktionaalinen, ja se osoittaa pelaajalle hyvin selkeästi miekaniskun kattaman etäisyyden. Zeldassa vauhtiviiva ei varsinaisesti edes seuraa miekkaa, mikä on varsin yleistä muissakin peleissä. Kritika, kuvassa 18, on referenssien ainut iOS-peli, ja siinä on menty varsin tyylitellyllä suunnalla. Toteutustapa hyvin samanlainen kuin Zeldassa, eli vauhtiviiva ei seuraa miekkaa ja vauhtiviiva on myös funktionaalinen.

Kuva 15. Heavenly Sword -pelin vauhtiviivaefekti (Realm Of Gaming - Heavenly Sword, Screenshots 2007).

Kuva 16. Vauhtiviivoja Ninja Gaiden 3 -pelissä (Ninja Gaiden 3 2012).

Kuva 17. Zelda – Hyrule Warriors (Zelda – Hyrule Warriors 2014).

Kuva 18. Kritika-pelin vauhtiviiva (Kritika 2014).

P2-pelissä miekan tai muun lyöntiaseen sivallus on pelkästään visuaalinen tehoste, jolla ei ole funktionaalista merkitystä. Ruutu on pieni, joten vauhtiviivan tulee olla selkeä Zeldan tapaan, mutta visuaalisesti Ninja Gaiden on lähempänä haluttua ilmettä. Heavenly Sword -pelin vauhtiviivat ovat kyllä hienot, mutta mobiililaitteelle tehoste on hieman turhan haalea. Zeldan ja Kritikan tavalla vauhtiviivan ei tarvitse seurata miekkaa, joten ei tarvitse tehdä mitään monimutkaista laskentaa esineen seuraamista varten. Tällöin vauhtiviivaa on hallittava jollain muulla tavoin, ja hyvin toden näköistä on, että vauhtiviivan data on hahmon animaatiossa, jolloin animaattorin on luotava data jossain animaatio-ohjelmassa. Seuraamalla miekkaa saadaan vauhtiviiva ilman, että kenenkään pitää varsinaisesti siitä tehdä, joskin se vaatii prosessointiaikaa.

3.5 Laavavirtaus

P2-pelissä on muun muassa demoninen susi, jonka silmistä pitäisi virrata laavaa. Kuvassa 19 on demonisen suden konseptikuva.

Kuva 19. P2-pelin demoninen susi.

World of Warcraftissa kiinnostava kohde on Lava Turtle, kuva 20, koska se on lähellä demonisen suden tarvitsemia tehosteita. World of Warcraftissa laavatehoste on kuitenkin hyvin yksinkertainen: hehkuva tekstuurikuva ja pari hehkupartikkelia. Silmissä oli kuitenkin oikeaa tunnelmaa. Tomb Raider Anniversary -pelissä, kuvassa 21, oli aiheeseen sopiva laavakenttä. Laavavirtaus oli varsin yksinkertainen lineaarinen UV-animaatio. UV-animaatio on helppo tapa, mutta toimii usein varsin hyvin, kunhan virtaussuunta pysyy samana. Left4Dead, kuvassa 22, ja Portal 2 käyttävät samaa vedenvirtaustekniikkaa, joka on varsin vaikuttavan näköinen tehoste. Siinä manipuloidaan veden ropoloisuustekstuuria virtaustekstuurin perusteella, mikä luo illuusion virtaavasta vedestä.

Kuva 20. World of Warcraftin Lava Turtle (Lava Turtle).

Kuva 21. Tomb Raider Anniversary -pelin virtaavaa laavaa (Tomb Raider Anniversary – Level 14).

Kuva 22. Valven Source-moottorin vesi (Vlachos 2010).

P2-pelissä laavatehoste on varsin pieni, ja se näkyy vain suden silmissä, joten monimutkaisten virtauskarttojen käyttö ei ole tarpeen. UV-animaatio ja tulipartikkelit riittäisivät luomaan illuusion tulisista silmistä ja niistä virtaavasta laavasta.

3.6 Häiriö maailmassa

Pelissä on häiriöitä, joihin kuljettaessa siirrytään taisteluun; ne ovat siis tietynlaisia portaaleja. Nämä häiriöt tulee indikoida pelaajalle hyvin selvästi, jotta pelaaja voi ottaa ne huomioon liikkuessaan pelimaailmassa.

Tutkituissa peleissä portaalit olivat ympyrämäisiä ja tasolla olevia tehosteita, kuten Oblivion- (kuva 23) tai Portal-pelissä (kuva 24). P2 on kuitenkin ruutupohjainen, ja siinä pelaaja liikkuu aina yhden ruudun verran johonkin suuntaan ja pelaajan pitäisi selvästi nähdä, missä ruudussa häiriö on, jotta hän osaa suunnitella reittinsä. Niinpä mukana on myös joitain ruutupohjaisia pelejä, kuten vanha UFO, kuvassa 25, ja suomalainen Grimrock, kuvassa 26. Niissä oli hyvin selvästi näkyvissä, missä ruudussa tehoste on, ja tehoste täyttää koko ruudun, mikä sopii paremmin P2-pelin tarpeisiin.

Kuva 23. Oblivion-pelin portaali (Oblivion Gate).

Kuva 24. Portal 2 (Portal 2).

Kuva 25. UFO: Enemy Unknown (X-Com Enemy Unknown VS. Aftermatch).

Kuva 26. Grimrock (They Play – Grimrock 2014).

4 Tehosteiden tekniikoita ja algoritmeja

4.1 Hiukkassimulaatio

Partikkelijärjestelmät eli hiukkasjärjestelmät pyrkivät parhaassa tapauksessa simuloimaan hyvin realistisesti oikean maailman hiukkasten käyttäytymistä, kuten kuvassa 27.

Kuva 27. Krakatoalla laskettu kuva Naidin nestesimulaatiosta.

Peleissä ei kuitenkaan saada laskettua tai edes piirrettyä riittävästi partikkeleita, jotta niissä voitaisiin oikeasti simuloida oikeaa maailmaa. Niinpä peleissä on tyytyminen pelkistettyihin versioihin, jotka näyttävät välttävästi oikean maailman ilmiöiltä. Illuusio oikeasta simulaatiosta usein hajoaa hyvinkin nopeasti, kun partikkelijärjestelmän kanssa tulisi jollain tapaa olla vuorovaikutuksessa. Asiaa kuitenkin helpottaa se, että tiedetään etukäteen, mitä rajoituksia on, ja voidaan välttää sellaisia tehosteita, joiden tekeminen olisi vaikeaa tai joille voidaan miettiä vaihtoehtoisia toteutustapoja. Useinkaan tehosteen ei ole tarpeen olla fysikaalisesti oikein, kunhan se näyttää hyvältä.

Partikkeli mielletään usein pieneksi pisteeksi, hiukkaseksi, jollainen se hiukkasfysiikassa toki onkin. Tietokonegrafiikassa partikkeli voi olla mitä tahansa ja minkäkokoinen tahansa. Usein se on quad-polygoni eli neliön mallinen polygoni, joka muodostuu kahdesta kolmiosta. Tämä polygoni on kuin mikä tahansa polygoni, joskin se usein piirretään hieman eri tavalla, mutta se on vain kolmioita siinä kuin muutkin polygonit. Hyvin yleisesti peleissä kaikki esineet muodostuvat kolmioista, mikä johtuu grafiikkakiihdytyksen tavasta rasteroida polygoneja, ja kolmio on muotona yksiselitteinen rasteroida. (Beam 2009.)

Partikkeli on yksinkertaisimmillaan vain transformaatiomatriisi, joka liikuttelee polygonia, kokonaista mallia, valoja tai mitä tahansa partikkelijärjestelmän halutaan liikuttelevan. Hankalaksi asian tekee fysiikka. Useimmiten kaikissa hiukkasjärjestelmissä on painovoima, joka siis vain vetää partikkeleita alaspäin annetulla kiihtyvyydellä. Tämän lisäksi voi olla tuulivoima, kitka, törmäystarkistus, pyörrevoima, työntövoima, ja niin edelleen. Lista vaihtelee käytetyn järjestelmän mukaan.

Unity käyttää partikkelijärjestelmää nimeltä Shuriken, joka on varsin kattava järjestelmä, mutta ei kovinkaan fysikaalinen. Siinä toki on painovoima, tuuli ja kevyt törmäystarkistuskin. Se ei kuitenkaan ole kovin vaikuttava fysiikan simulointityökaluna. Järjestelmän etu on muokattavuus, minkä vuoksi sillä voidaan tehdä paljon hyvin erityyppisiä efektejä. Kuvassa 28 on Shuriken järjestelmällä luotu savu. Unity tarjoaa järjestelmälleen myös kattavan API:n, jonka avulla voi halutessaan kirjoittaa vaikka oman fysiikkamallinnuksen Shurikenin päälle. (Unity Manual – Particle System.)

Kuva 28. Unityn Shuriken-partikkelijärjestelmällä tehty savutehoste (Unity Manual – Particle System).

4.2 Varjostimet

Nykypeleissä varjostimet (engl. Shaders) ovat arkipäivää, ja kaikki mitä voidaan, tehdään varjostimilla. Kuvassa 29 on esimerkki ropoloisuusvarjostimesta.

Kuva 29. Esimerkki Unityn ropoloisuusvarjostimesta. Tehty Unityn Surface-varjostimella. (Unity, Unity Manual - Shader Reference, n.d.)

Varjostimien käyttö on suosittua, koska se säästää prosessoriaikaa (CPU), sillä varjostinkoodi suoritetaan grafiikkaprosessorissa (GPU). Myös vähänkin uudemmassa mobiililaitteessa on grafiikkaprosessori, ja pöytäkoneessa tai kannettavassa tietokoneessa grafiikkaprosessori on itsestäänselvyys. Eri grafiikkaprosessorien tehoissa ja ominaisuuksissa saattaa olla merkittäviä eroja, joten peliä tehdessä varjostimet tulee aina testata kehnoimmalla tuetulla laitteistolla. (Lammers 2013.)

Unity tukee varjostimia monipuolisesti. Unityssa voidaan kirjoittaa HLSL-, GLSL- ja CG-varjostimia. Niiden lisäksi voidaan kirjoittaa Unityn omia Surface-varjostimia, jotka oikeasti ovat kuitenkin CG- tai HLSL-varjostimia, riippuen siitä mille, alustalle peliä käännetään. Kaikki kolme varjostinkieltä tukevat kärkipistevarjostimia ja pikselivarjostimia. (Unity Manual - Shader Reference.)

Kärkipistevarjostin

Kärkipistevarjostin (engl. Vertex Shader) on koodi, joka suoritetaan jokaiselle kärkipisteelle joka kerta, kun grafiikkaprosessori alkaa prosessoida niitä. Varjostimessa usein vähintään projisoidaan kärkipisteet haluttuun perspektiiviprojektioon. Niissä voidaan kuitenkin tehdä kärkipisteelle melkeinpä mitä tahansa. Ongelmana on, että pelikoodi ei tiedä, mitä varjostin tekee, koska varjostinkoodi suoritetaan pelikoodin jälkeen. Jos varjostin siirtää kärkipistettä, peli kuitenkin luulee, että kärkipiste on edelleen siellä, missä ennenkin. (Lammers 2013.)

Pikselivarjostin

Pikselivarjostin (engl. Fragment Shader) suoritetaan polygonin jokaiselle ”pikselille”. Pikseli ei kuitenkaan ole aivan oikea termi, sillä kyseessä on piste polygonissa eikä sillä kuvata niinkään ruudulla olevaa pikseliä, joka lopputuloksena kylläkin piirretään. Jokaiselle polygonin pikselille suoritetaan tässä varjostimessa oleva koodi. Tämän koodin kanssa tulee olla erityisen varovainen, koska se suoritetaan hyvin monta kertaa kuvaa laskettaessa. Erikoisluontoista pikselivarjostimessa ovat muuttujat, jotka interpoloituvat automaattisesti, kunhan vain kärkipistevarjostin antaa muuttujille arvot, joiden välillä interpoloida. Tämän ominaisuuden avulla voidaan tehdä esimerkiksi teksturointi tai ropoloisuusteksturointi. (Lammers 2013.)

4.3 Vauhtiviivatekniikat

Tail Arc Renderer

Nick Gronow on kirjoittanut Unity-moottoriin Tail Arc Renderer -laajennuksen, joka luo oman vauhtiviivaobjektinsa. Gronow’n tapa piirtää objektia annettujen pisteiden välille ja kääntää mallin aina kameraan päin. Suurin ongelma hänen tavassaan on, että objektin generointiajat määritellään käsin syötettyinä sekuntiarvoina. Tästä seuraa, että tulee tietää etukäteen, kuinka kauan vauhtiviiva näkyy. Keston antaminen sekunteina on myös ikävän työlästä ja vaatii helposti jopa laskemista, jos vauhtiviiva ei alakaan animaation alusta. Sekuntiaikaa ei myöskään voi suoraan tallentaa animaation mukaan, vaan se vaatisi jonkin oman tavan, jolla kertoa vauhtiviivalle, miten kauan tämän animaation kohdalla vauhtiviivan tulee näkyä. (Gronow 2012.)

Unity Trail Renderer

Unity-moottori itsessäänkin tarjoaa vauhtiviivajärjestelmän. Tämä järjestelmä ei kuitenkaan ole kovin hyvin kontrolloitavissa. Siinä voi kyllä määrittää vauhtiviivan leveyden, mutta se perustuu annettuun vakioarvoon, eli vauhtiviivaa ei pysty animaation aikana leventämään tai kutistamaan animaattorin toivomalla tavalla. Trail renderer lisäksi kärsii samoista ongelmista kuin Gronow’n tapa. (Unity Manual - Tail Renderer.)

4.4 Nesteenvirtaustekniikat

UV-vieritys

UV-vieritystekniikan englanninkielinen nimi on UV-Scrolling. Se on hyvin yksinkertainen ja halpa tapa tehdä virtaavaa nestettä. Se kuitenkin toimii monessa tapauksessa kohtuullisesti, mutta sillä on vakavia rajoituksia.

Perusajatus on, että tekstuurikuva on saumattomasti monistuva, ks. kuva 30, jolloin voidaan toistaa samaa kuvaa eikä saumaa näy. Näin voidaan pyörittää tekstuuriavaruutta ympäri ja saada aikaan liikettä, joka tapahtuu vain tekstuurissa eikä objektin geometrinen data muutu.

Kuva 30. Saumattomasti toistuva kuva (CZE).

Algoritmille annetaan parametrit suunta $\vec{D}$ ja nopeus v. Kappaleen jokaiselle pisteelle lasketaan uudet UV-koordinaatit kaavalla, jossa $\overrightarrow{P_{uv}}$ on kärkipisteen UV-koordinaatit ja t kulunut aika:

$$\overrightarrow{UV} = \overrightarrow{P_{uv}} + \vec{D} * v * t$$

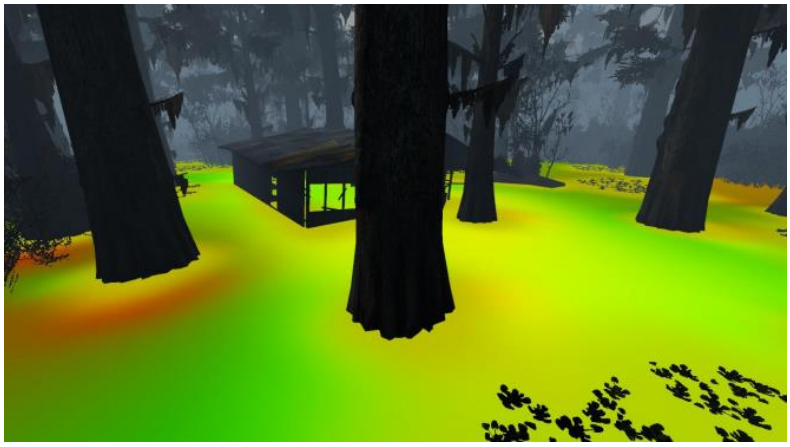
Suuntavektori $\vec{D}$ skaalataan nopeudella v ja lisätään senhetkiseen UV-koordinaattiin. Nopeusarvoa skaalataan ajalla, mikä tässä tarkoittaa kulunutta aikaa sitten viime päivi-

tyksen. Jos tätä aikakerrointa ei ole, efektin nopeus vaihtelee riippuen päivitysnopeudesta, eli nopeilla tietokoneilla efekti liikkuisi nopeammin ja siksi tarvitaan aikakerroin pitämään efektin visuaalinen ilme samana riippumatta päivitysnopeudesta.

Tapa toimii hyvin, kunhan virtaus on nopeudeltaan aina sama. Jos virtausnopeutta halutaan vaihtaa, se vaikuttaa kerralla koko virtaukseen eikä vain yhteen kohtaan. Lisäksi virtaus luottaa siihen, että mallilla, jolla virtaus on, on UV-kartta, joka on virtauksen suuntainen. UV-kartta määrää, minne virtaus menee, eikä virtauksen suuntaan voi muulla tavoin vaikuttaa. Niinpä virtaus on aina joko eteen- tai taaksepäin UV-kartan mukaan, eikä virtaus voi esimerkiksi kiertää estettä tai reagoida siihen millään tavalla. Tapa ei siis ole kovin hyvä kuvaamaan veden virtausta isoilla alueilla. (Lammers 2013.)

Virtauskartta

Virtauskartta on Valven kehittämä tekniikka, jossa liikutetaan ropoloisuuskarttaa luomaan illuusio virtaavasta vedestä. Toisin kuin tavanomaisessa UV-avaruuden liikuttamisessa, tässä tekniikassa liikkeen suunta ja nopeus kirjoitetaan tekstuurikuvaan, ks. kuva 31, jota kutsutaan virtauskartaksi (engl. Flowmap). Ajatus virtauskartoista ei ole alun perin Valven vaan perustuu Nelson Maxin ja Barry Beckerin kirjoittamaan tutkielmaan Flow Visualization Using Moving Textures,1995.



Kuva 31. Vedenpinta teksturoitu virtauskartalla (Vlachos 2010).

Taiteilija luo virtauskartan joko piirtämällä sellaisen käsin tai käyttämällä jotain käytössä olevaa ”vektorivoima-alue”-työkalua. Valve käytti tähän Sidefx Houdini -ohjelmistoa. Kun

virtauskartta on luotu, voidaan sen avulla venyttää tekstuuria. Värit virtauskartassa edustavat suuntia. Punainen venyttää tekstuuria U-suunnassa ja vihreä V-suunnassa. Kun väri on arvossa 0,5 (tai 128 8-bittisessä kanavassa), venytystä ei tapahdu. Esimerkkinä oletetaan, että punainen väri on 0,2 ja vihreä 0,7, jolloin liikevektori $\vec{V}$ on

$$\vec{V} = ((R - 0.5) * 2, (G - 0.5) * 2)$$
$$\vec{V} = ((0.2 * 0.5) * 2, (0.7 - 0.5) * 2$$
$$\vec{V} = (-0.3 * 2, 0.2 * 2)$$
$$\vec{V} = (-0.6, 0.4).$$

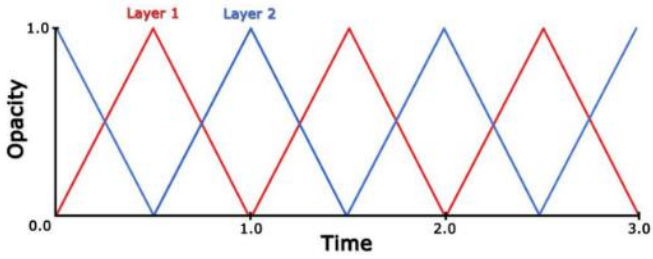
Nyt voidaan käyttää tätä liikevektoria manipuloimaan tekstuuria. Jokaiselle tekstuurin pisteelle lasketaan

$$\vec{P} = \overrightarrow{UV} * \vec{V},$$

jossa $\vec{P}$ on uusi UV-piste ja $\overrightarrow{UV}$ on nykyinen UV-piste. Tämä luo tekstuuriin vääristymää, joka mukailee virtauskarttaa. Jos yllä olevaan kaavaan lisätään aika, saadaan animoituva vääristymä:

$$\vec{P} = \overrightarrow{UV} * \vec{V} * t.$$

Tästä seuraa ongelmana se, että tekstuuri alkaa vääristyä liikaa, koska kulunut aika vääristää tekstuuria aina vain enemmän. Ongelma korjataan käyttämällä kahta tekstuuria. Kun toinen alkaa vääristyä liikaa, toinen tuodaan vanhan päälle sitä mukaa, kuin vääristymä lisääntyy. Lopulta uusi tekstuuri alkaa vääristyä liikaa, jolloin vanha tekstuuri palautetaan takaisin alkutilaan ja tuodaan uuden tekstuurin päälle sitä mukaa, kuin uusi tekstuuri alkaa vääristyä liikaa. Kuvassa 32 näkyy, missä vaiheissa tekstuureja joko tuodaan esille tai häivytetään pois. Tätä jatketaan ikuisesti, ja saadaan aikaan illuusio virtauksesta.



Kuva 32. Käyrät osoittavat, miten kahta tekstuuria tuodaan esille ja pois. Tässä tekstuuri tuodaan esiin ja pois kerran yhdessä kierrossa.

Tekniikassa on joitakin ongelmia. Vääristymä ei saa olla kovin suurta, jotta efekti edelleen näyttää hyvältä. Ropoloisuustekstuurien jaksoittaisesta käytöstä aiheutuu toistuvuutta, jota Valve korjaa lisäämällä animaatioon satunnaisuutta. (Vlachos 2010.)

4.5 Tekstuurisarja-animaatio

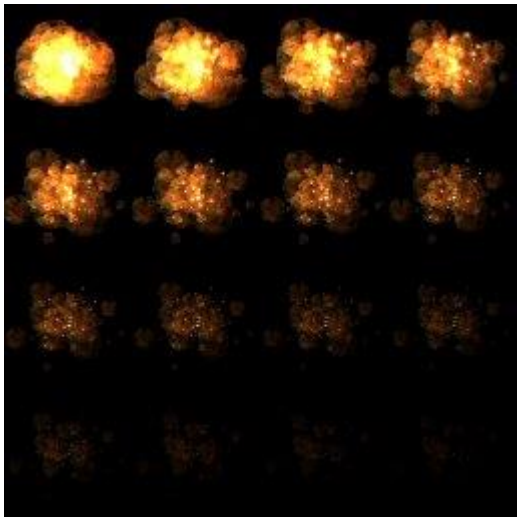
Kun käytetään tekstuuria, joka on animoitu, on suotavaa, ettei tekstuuria tarvitse vaihtaa. Tekstuurikuvan vaihtaminen on kallista, koska Unity tekee sisäisesti grafiikkakiihdyttimelle komennon asettaakseen uuden tekstuurikuvan. Tämä hidastaa grafiikkaprosessorin toimintaa, joten on nopeampaa laittaa kaikki kuvat animaatiossa samaan tekstuurikuvaan ja liikutella mallin UV-koordinaatteja sopivissa askeleissa. Tätä tekniikkaa kutsutaan nimellä Texture Sheet Animation eli tekstuurisarja-animaatio. (Wloka 2009.)

Tekniikka on varsin vanha, sitä on käytetty jo todella vanhoissa kaksiulotteisissa peleissä, kuten esimerkiksi Monkey Island, ks. kuva 33.

Kuva 33. Monkey Island -pelin päähahmon animaatiot yhdessä kuvassa (Monkey Island 1990).

Monkey Islandin aikaan pohjimmainen syy tehdä tekstuurisarja-animaatiota on ollut muisti. Yksi isompi kuva vie vähemmän muistia kuin monta pientä. Eri syistä sama tekniikka on edelleen tarpeen. Nykypeleissä monet, yleensä pienet yksityiskohdat, kuten vilkkuva valo jossain ohjainpöydällä tai jokin ruutu jossain laitteessa, usein toimivat tekstuurisarjakuvalla. Esimerkiksi ensimmäisen Max Payne -pelin sisäiset ”tv-ohjelmat” on tehty tällä tekniikalla.

Jotta voidaan laskea sopiva askel UV-koordinaattien siirtoon, pitää ensin tietää, montako kuvaa kuvassa on, rivissä ja sarakkeissa. Oletetaan kuvan 34 kaltainen tilanne.



Kuva 34. Positech Gamesin Explosion Generator -työkalulla generoitu kuvasarja-animaatio (Positech Games).

Kuvassa on siis 4 kuvaa rivissä ja 4 kuvaa sarakkeissa eli yhteensä 16 kuvaa:

$$c_x = 4$$
$$c_y = 4.$$

Seuraavaksi lasketaan, mikä kuva tulee näyttää missäkin ajan hetkessä. Jotta tämä voidaan laskea, pitää ensin päättää, millä nopeudella kuvia näytetään. Tavallisesti nopeus annetaan muodossa kuvia sekunnissa (fps), ja yleisesti suosittu nopeus on 30 kuvaa sekunnissa:

$$f = 30,$$

jolloin yhden kuvan kestoksi tulee

$$s = \frac{1s}{30} * 1000 = 33.3333ms.$$

Kun otetaan aika (t), joka Unitysta tulee sekunteina, se pitää ensin muuntaa millisekunneiksi:

$$t_m = t * 1000ms,$$

ja voidaan laskea, missä kohtaa animaatiota ollaan millisekunneissa (p). Lopuksi lasketaan kyseisen kohdan kuvan järjestysnumero (i) koko kuvasarjassa:

$$p = t_m \left(mod\left(c_x * c_y * s\right)\right)$$
$$i = \left\lfloor \frac{p}{s} \right\rfloor.$$

Kun järjestysnumero koko kuvasarjassa on tiedossa, voidaan laskea järjestysnumero riveissä ja sarakkeissa:

$$y = \left\lfloor \frac{i}{c_x} \right\rfloor$$
$$x = i\left(mod\left(c_x * c_y\right)\right).$$

Lopulta voidaan laskea UV-koordinaatit, ja koska toimimme tekstuuriavaruudessa, halutaan arvot välille 0 ja 1, eli normalisoidaan järjestysnumerot:

$$u = \frac{x}{c_x}$$
$$v = \frac{y}{c_y}.$$

Nyt voidaan näyttää aina sopivaa kohtaa tekstuuriavaruudesta, ja tekstuuri näyttää vaihtuvan aina uuteen kuvaan, vaikkakin todellisuudessa UV-koordinaatteja vain siirretään. (Lamothe 1999.)

4.6 Optimointi

Pelin tapahtumia ja grafiikkaa lasketaan reaaliaikaisesti, ja pelin tulee näyttää noin 30 kuvaa sekunnissa, jotta se edelleen näyttää sulavalta eikä ala näkyvästi nytkähdellä. Tämä tarkoittaa sitä, että pelillä on 33,33 millisekuntia aikaa laskea ja piirtää uusi kuva. Tästä syystä kaiken pitää tapahtua hyvin nopeasti. Nopeuden lisäksi pelillä on myös muistirajoitteita. Laitteessa, jolla peliä pelataan, on rajallinen määrä muistia, ja kaikki pelin tarvitsemat mediat, kuten kuvat, 3D-mallit, äänet, ynnä muuta sellaista. pitää ladata muistiin, jotta mediaa voidaan käyttää. Muistin määrä tulee huomioida, jotta se ei lopu pelin aikana kesken. Muistin loppuminen johtaa ohjelmavirheeseen ja käyttöjärjestelmä suojaa itsensä lopettamalla pelin. Pahimmassa tapauksessa koko käyttöjärjestelmä jumiutuu, mutta parhaassakin tapauksessa peli sammuu hallitsemattomasti ja pelistä saattaa kadota tietoa, kuten senhetkinen pelitilanne. (Alhman 2014.)

Nopeuden optimointi

Nykyisissä laitteissa on keskusprosessori ja grafiikkaprosessori ja kumpiakin on usein enemmän kuin yksi. Unity hoitaa sisäisesti paljonkin säikeistämistä, mikä on tärkeää moniytimisten prosessorien kanssa. Säikeet ovat ohjelmakoodia, joka suoritetaan prosessorin ytimessä, ja säikeillä ytimiä voidaan käyttää yhtä aikaa ja asioita voidaan laskea rinnakkain. Säikeistäkin huolimatta on helppoa saada pelistä hidas tehottomalla koodilla. (Alhman 2014.)

Eri alustat tekevät eri asioita eri tavoin, ja toiset alustat tekevät jotkut asiat nopeammin kuin toiset. Niinpä on tärkeää, että tietää, mitkä asiat milläkin alustalla ovat hitaita tai nopeita. On myös tärkeää tietää, mikä prosessori mitäkin asiaa suorittaa. Voi olla tilanne, jossa grafiikkaprosessori tekee paljon laskentaa, mutta keskusprosessori ei tee juuri mitään. Tällöin peli voisi hyvin tehdä jotain keskusprosessorilla, kunhan ei lisää kuormaa grafiikkaprosessorille. Myös se, mitä tehdään koodissa, aiheuttaa hyvin erilaisia operaatioita, joista toiset ovat hitaampia kuin toiset. Esimerkiksi grafiikkaprosessori on hyvä suorittamaan matriisi- ja vektorilaskuja, mutta huono tekemään ehtolauseita, vaikka se onkin siinä mahdollista. (Alhman 2014.)

Muistinhallinta on tärkeää, myös kielissä, jotka tekevät automaattista muistinhallintaa, kuten C#. Vaikka modernissa ohjelmointikielessä ei tarvitse murehtia muistivuodoista,

kuten esimerkiksi C- tai C++-kielessä, ei muistinhallintaa voi silti unohtaa. Muistin varaaminen vie prosessoriaikaa, ja tietynlainen varaaminen maksaa enemmän kuin toinen. (Alhman 2014.)

Muistia on kahta tyyppiä, pino (stack) ja keko (heap). Näistä pino on nopeaa käyttää, ja prosessori tyhjentää sen automaattisesti käytön jälkeen. Pino on kuitenkin käytettävissä vain sen funktion sisällä, jossa se luodaan. Keko taas on käytettävissä missä tahansa ohjelman sisällä, mutta sen käyttö on hitaampaa. C# tyhjentää tämänkin muistin automaattisesti, mutta muistintyhjennysprosessi on hidas ja saattaa aiheuttaa nytkähdyksen, kun se tapahtuu. Muistintyhjentäminen on automaattista, eikä mitenkään helposti voi hallita, milloin se tapahtuu. On siis tärkeää, ettei peli varaa keko muistia useasti. Parhaassa tapauksessa kaikki kekomuisti varataan pelin alussa ja sen jälkeen käytetään vain pinomuistia. (Alhman 2014.)

C#- tai JavaScript-kielissä ei ole mitenkään suoraan selvää, mitä muistia mikäkin asia varaa. JavaScript varsinkin on ongelmallinen kieli, koska se käsittää funktiotkin olioina, jotka varaavat kekomuistia. Tämä tarkoittaa sitä, että jokainen funktio tekee hitaan muistivarauksen, joka pitää jossain vaiheessa vapauttaa. Tämä on todella haitallista pelin tehokkuudelle. Unity taas tekee kirjoitetun koodin päälle omaa sisäistä optimointia, joka saattaa korjata joitain ongelmia. Tästä toki seuraa, että pitää tietää, mitä Unity koodille tekee, jotta sille taas osataan kirjoittaa nopeaa koodia. C#-kielessä tilanne on vähän helpompi, sillä siinä oliot ovat oikeasti olioita. C# tukee myös rakenteita, jotka ovat paljon olioiden kaltaisia, mutta eivät varaa kekomuistia. Tästä syystä muun muassa Unityn Vector-rakenteet ovat nimenomaan rakenteita eivätkä olioita. (Alhman 2014.)

Aina ei kuitenkaan ole aivan selvää, mikä on olio. Esimerkiksi C#-kielessä taulukko on itse asiassa olio, joka siis varaa kekomuistia (kuva 35). Tässä onkin yksi tärkeä optimointikohta. Taulukon pitäisi aina olla luokan muuttuja, joka on alustettu johonkin kokoon, minkä jälkeen taulukkoa ei luoda uudestaan vaan tyhjennetään (kuva 36). Tällä varmistetaan, että muisti on varattu vain kerran. (Alhman 2014.)

```
1  public class Luokka
2  {
3      public void Funktio()
4      {
5          int[] taulukko = new int[1000];
6      }
7  }
```

Kuva 35. Tässä koodissa taulukko luodaan väärin.

```
1  public class Luokka
2  {
3      private int[] taulukko = null;
4      const int MAARA = 1000;
5
6      public Luokka()
7      {
8          taulukko = new int[MAARA];
9      }
10
11     public void Funktio()
12     {
13         // Täytä taulukko jollain.
14     }
15
16     public void Tyhjenna()
17     {
18         for (int i=0; i < MAARA; ++i)
19             taulukko[i] = 0;
20     }
21 }
```

Kuva 36. Tässä koodissa taulukkoa käytetään oikein.

Tämä tapa on nopeampaa, mutta se myös auttaa arvioimaan muistin kokonaiskulutusta. Ongelma toki on, että nopeudelle optimoitu koodi ei tarkoita, että koodia olisi vähemmän. Hyvin usein asia on juuri toisin päin. On myös hyvä vältellä kyselykieliä, kuten LINQ. Ne kyllä suurimmalta osaltaan toimivat, mutta tekevät paljon enemmän operaatioita koodin takana, kuin on suoraan nähtävissä. Näihin prosesseihin ei myöskään voi mitenkään itse vaikuttaa, joten on parempi vain kirjoittaa hakurutiinit itse, jotta tietää, mitä ne tekevät ja niitä voi tarpeen vaatiessa optimoida. (Alhman 2014.)

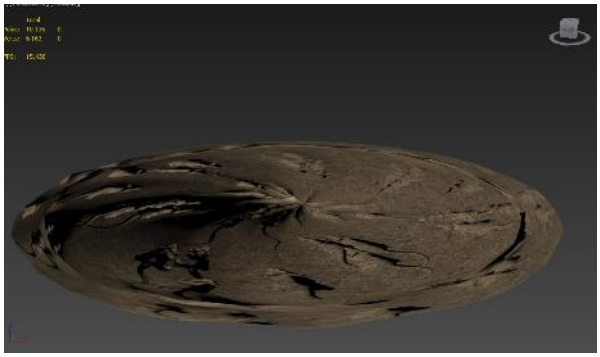
Muistin optimointi

Kun puhutaan muistin optimoinnista, ei välttämättä tarkoiteta pino- tai kekomuistia vaan muistin kokonaiskulutuksen alentamista ja datan määrää. Muistia on monessa paikassa: laitteen keskusmuisti, grafiikkakiihdyttimen muisti ja massamuisti. Keskusmuistiin ladataan kaikki data, josta se on prosessorin käytettävissä. Osa datasta kuitenkin siirretään grafiikkakiihdyttimille, jotta grafiikkaprosessori saa tarvitsemansa datan. Data muistissa on binääridataa, jolla on jokin koko. Esimerkiksi yksi kokonaislukuarvo on usein int-tyyppinen muuttuja, jolla on kokoa 32 bittiä. Se siis vie muistia 32 bittiä eli 4 tavua, niin kauan kuin kyseinen muuttuja on muistissa. Bittimäärä vaikuttaa siihen, montako numeroa muuttujalla voidaan esittää. 32-bittisellä muuttujalla numeroavaruus on -2 147 483 648 … 2 147 483 647. Jos ei kuitenkaan tarvita näin suurta numeroavaruutta, voidaan int-tyyppi vaihtaa vaikka short-tyypiksi. Short on muutoin samanlainen, mutta se on 16-bittinen, eli se voi tallentaa vain arvot välillä -32 768 … 32 767. Short kuitenkin vie puolet vähemmän muistia kuin int. Yksittäisessä tapauksessa tällä ei ole juuri väliä, mutta jos tallennetaan vaikka kuvan väriarvoja ja kuvassa on useita kymmeniä tuhansia pikseleitä, ero alkaa näkyä, puhumattakaan videosta, jossa kuvia on useita tuhansia. (MSDN: int 2013.)

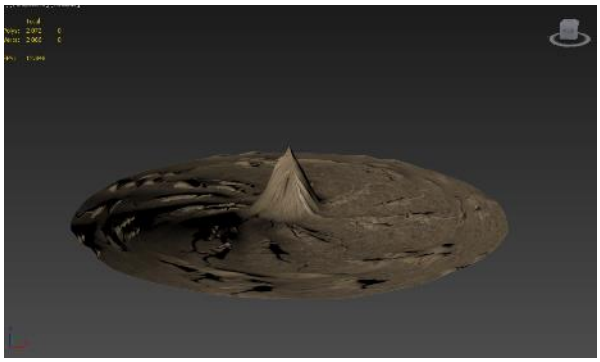
Nykypelissä on usein erittäin paljon dataa, joka vie tilaa massamuistista, koska kaikki data pitää tallentaa johonkin. Massamuisti on hidasta muistia, mutta samalla myös halpaa muistia, ja massamuistissa oleva data säilyy siellä myös laitteen uudelleen käynnistyttyä. Suuri datamäärä ei kerralla mahdu keskusmuistiin, joten dataa on ladattava osissa, siksi monet pelit on jaettu eri kenttiin. Jotkut pelit myös lataavat dataa kaiken aikaa, jolloin kauas jääneet asiat poistetaan muistista ja ladataan muistiin lähemmäs tulevia asioita. Suuri datamäärä on ongelma, sillä sen varastointi vie paljon tilaa ja massamuistia on aina rajallisesti. Iso datamäärä myös hankaloittaa pelin levittämistä. Jotkut laitteet käyttävät DVD-levyjä pelien levitykseen, ja DVD:n tila nykypelille on varsin pieni. Toiset laitteet käyttävät BlueRay-levyjä, joissa on vielä nykyisin tarpeeksi tilaa, mutta BlueRay-levyn hitauden vuoksi suurin osa datasta on kopioitava kiintolevylle, jota ei yleensä ole 500 gigatavua enempää, mikä tarkoittaa, että kerralla voi olla asennettuna vajaa kymmenen peliä. Mobiilissa pelit ladataan poikkeuksetta verkosta, jossa latausaika on ongelma. Mobiililaitteen massamuisti on myös hyvin yleisesti todella pieni ja isompia 2 – 5 gigatavun pelejä ei montaa voi asentaa. Isoilla peleillä latausaika kasvaa helposti varsin pitkäksi. Mobiilissa tunnin latausaika ei ole mitenkään harvinainen, mutta konsolilla uuden 50 gigatavun pelin lataaminen saattaa kestää koko päivän. (Alhman 2014.)

Tilantarvetta voidaan vähentää monella tapaa, ja usein se myös vähentää keskusmuistin tarvetta. Tekstuuriresoluutio on yleensä ensimmäisiä asioita, joilla ongelmaan voidaan vaikuttaa. Laskemalla resoluutio puoleen voidaan säästää 75 % tekstuurien vaatimasta muistista. Tämä tietysti laskee myös tekstuurien laatua. Äänien tasoa voidaan laskea tai äänet voidaan muuntaa monoääniksi. Videoiden pakkaaminen on erityisen tärkeää. Pelissä on usein aivan omia videoformaatteja, kuten bink-video. Mobiilissa, jossa tilarajoitukset ovat erityisen tärkeitä, on tarpeen pakata dataa niin paljon kuin mahdollista. Esimerkiksi tekstuuridata: usein tekstuuridata on enemmän kuin väritekstuurikuva. Tavallisesti siihen kuuluu myös ropoloisuuskuva ja valon heijastuskuva, joskus myös hehkukuvia, heijastuskuvia, korkeuskuvia, ynnä muuta sellaista. Kaikki nämä kuvat eivät sisällä varsinaisesti kuvaa, jota katsotaan, vaan dataa, jota käytetään varjostimessa. Datan syöttäminen varjostimelle tekstuurina on varsin helppoa, ja siksi dataa pakataan mieluiten kuvaan. Kuva on tietokoneelle vain dataa, jota prosessoidaan sovitulla tavalla, ja tietokone osaa piirtää sen. Piirtoprosessia ei kuitenkaan tarvitse tehdä millään sovitulla tavalla, vaan data voidaan käsitellä, miten tilanteeseen parhaiten sopii. (Alhman 2014.)

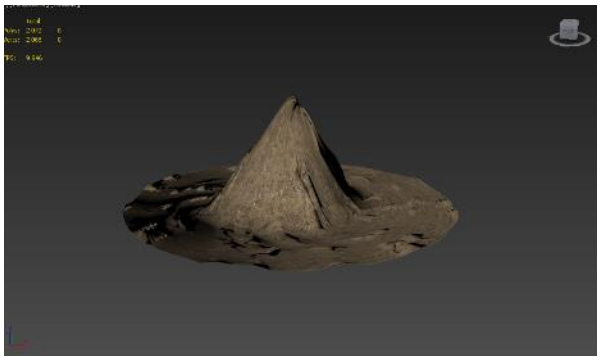
Tarkastellaan esimerkkinä tilannetta, jossa on vaikeasti animoituva objekti, kuten kuvien 37 – 41 sarjassa



Kuva 37. Objektin animaation ensimmäinen ruutu.



Kuva 38. Animaatio pienen hetken kuluttua.

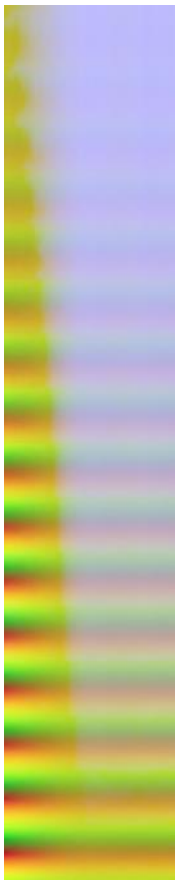


Kuva 39. Animaatio puolivälissä.

Kuva 40. Animaatio lähellä loppua.

Kuva 41. Animaation viimeinen ruutu.

Tilanne on vaikea esittää hyvin luilla, koska objekti on yhdessä vaiheessa aivan lituska ja muodostuu siitä omaan muotoonsa. Luilla on hyvä esittää objekteja, joiden yleinen muoto säilyy samana ja muotoa vain käännellään, kuten ihmisen animaatio. Toki esimerkin tilanteenkin voisi tehdä luilla käyttämällä todella runsaan määrän luita, mutta luiden koko ajatus on yksinkertaistaa mallia, jotta sitä voidaan animoida. Esimerkin tilanteessa animaatio on tuotettu 3dsMaxissa Displace-, Melt- ja Noise-muuntimilla. Esimerkin tilanne on kehitteillä olevasta hahmosta, jonka tulee muodostua maasta ja luoda itselleen ensin ”kotilo”, joka sitten räjähtää kappaleiksi hahmon hypätessä sen sisältä ulos. Kyseessä on siis vain yksi hetki, joka näkyy kerran, kun hahmo tulee ruudulle. Sen ruutuaika on niin lyhyt, ettei sille ole mielekästä kuluttaa muistia sitä määrää, mitä vaadittaisiin, jos jokainen objektin kärkipisteen animaatio tallennettaisiin muistiin tai sille luotaisiin riittävästi luita. Niinpä tilanne käyttää tapaa, jota voitaisiin kutsua nimellä Animated Vertex Map (animoitu kärkipistekartta). Kärkipisteiden liike tallennetaan 24-bittiseen kuvaan, jotta se olisi helppoa syöttää grafiikkakiihdyttimeen. Ensin analysoidaan liikkeen vaatima suurin tilavuus animaation aikana, minkä jälkeen jokainen kärkipiste normalisoidaan tällä tilavuudella jokaisessa animaation kuvassa. Arvot tallennetaan kuvaan, jokainen kärkipiste omalle rivilleen (Y-akseli) ja jokainen animaatioruutu omalle sarakkeelleen (X-akseli). Tästä syntyy kuvan 42 mukainen kuva.



Kuva 42. Kuva on laskettu 3dsMaxissa Maxscript-kielen avulla analysoimalla animaatiota.

Esimerkissä on ylöspäin nousevaa liikettä, joka näkyy kartassa sinisenä. Muut värit kuvastavat liikettä X- ja Z-suunnissa. Tämä kuva voidaan ladata GPU:n muistiin ja laskea kärkipisteiden paikat uudestaan skaalaamalla normalisoidut väriarvot animaation tilavuudella. Huomioitavaa tässä on se, että data on 8 bittiä akselia kohden. Se on varsin vähän kuvastamaan kolmiulotteista paikkaa, siitäkin huolimatta, että se on normalisoitu vain animaation tilavuuteen. Tästä aiheutuu epätarkkuutta, mutta liike kuitenkin pääosin säilyy ja se voidaan laskea kokonaan GPU:lla. Muistia tarvitaan animaatiokartan verran, jonka koko on

$$s = c * f * 8 * 3.$$

Kaavassa s on koko bitteinä, c on kärkipisteiden määrä, f on animaatioruutujen määrä, joka kerrotaan 8:lla, koska data on 8-bittistä, ja kerrotaan vielä 3:lla, koska värikanavia on käytössä kolme, R, G ja B. Tarkkuutta voisi vielä kasvattaa ottamalla alpha-kanavan mukaan. Esimerkissä kärkipisteitä on 1025 ja animaatioruutuja 200. Muistia tarvitaan,

$$s = 1025 * 200 * 8 * 3 = 4920000\ bittiä\ eli\ 615000\ tavua\ eli\ 615\ kilobittiä.$$

Tallennettuna PNG-kuvana tilaa vaaditaan levyltä 197 kilobittiä PNG-kuvan pakkausalgoritmin vuoksi. PNG käyttää Deflate-pakkausalgoritmia, joka on sama, jota esimerkiksi Zip-pakkaus käyttää. (Bits'n'Bites 2011.)

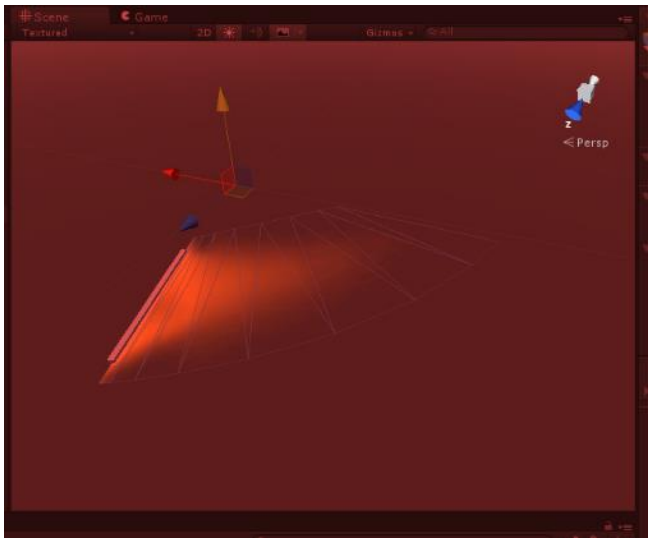
5 P2-pelin tehosteet

5.1 Vauhtiviivat

P2-peli tarvitsee vauhtiviivoja miekan sivalluksiin ja luoteihin. Hiukkaspohjaistaratkaisua tutkittiin, mutta ongelmaksi muodostui, että hiukkaset saattoivat jättää vauhtiviivaan eitoivottuja rakoja eivätkä ne luoneet yhtenäistä pintaa. Partikkeleja on myös taiteellisesti hankala ohjata. Tapa, joka loisi tasaisesti täytettyä pintaa, olisi tarpeeseen sopivampi.

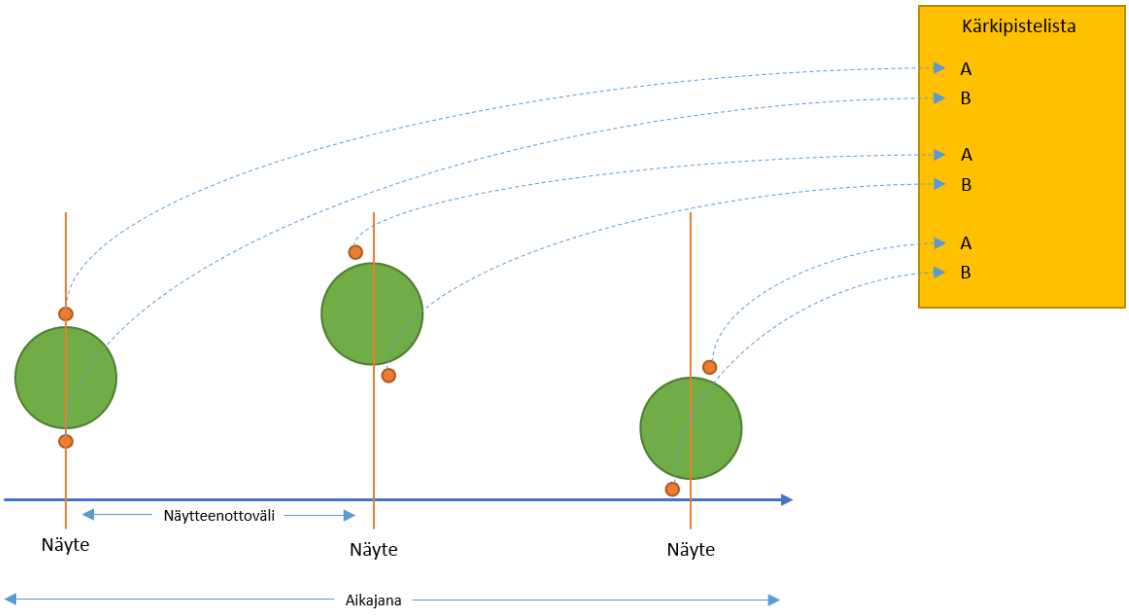
Järjestelmän nimeksi tuli Vauhtiviivaobjekti (engl. Trail Mesh). Siinä vauhtiviivaobjekti generoidaan kahden pisteen välille, joista otetaan annetulla aikajaksolla näytteitä annetulla tiheydellä. Joka näytteenottokerralla uudet pisteet lisätään vauhtiviivamalliin, ja näin muodostetaan malli, joka seuraa liikkuvaa kohdetta, kuten miekka tai luoti.

Kuvassa 44 vihreä pallo kuvastaa objektia, jolle halutaan piirtää vauhtiviivat. Pienet keltaiset pallot ovat pisteet, joiden välille vauhtiviivaobjekti luodaan. Ylempi piste on A ja alempi B. Pisteet määritellään käsin, eikä mikään siis automaattisesti laske objektin kokoa. Näin vauhtiviivaobjektin ulkoasua voidaan paremmin hallita.



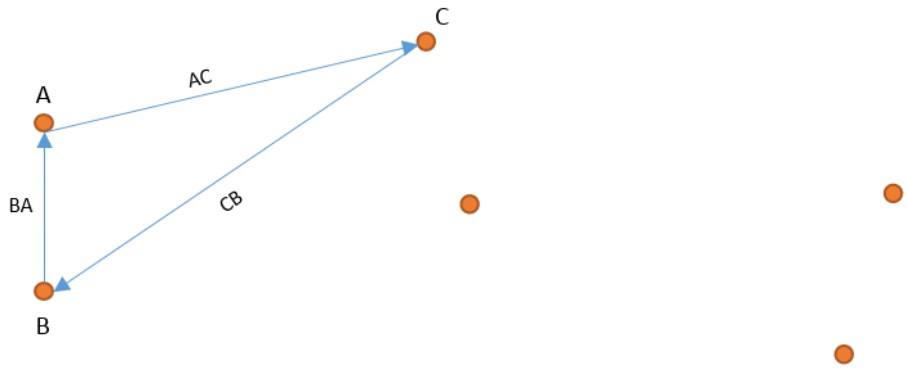
Kuva 43. P2-pelin vauhtiviivaobjekti generoidaan ajonaikaisesti seuraamalla kahta pistettä.

Jokaisella näytteenottokerralla pisteiden A ja B paikka maailmassa luetaan ja tallennetaan listaan. Pisteet tallennetaan listaan järjestyksessä A ja sitten B ja vanhemmat pisteet ensin (kuva 44).



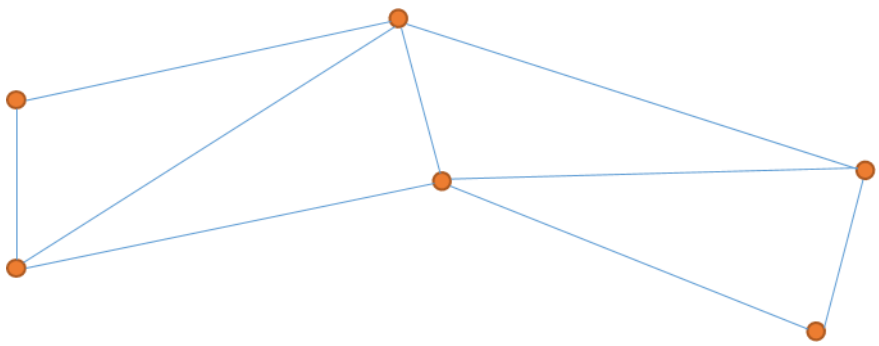
Kuva 44. Pisteistä A ja B otetaan näyte tietyllä näytteenottovälillä.

Koska pisteiden järjestys listassa tiedetään, voidaan pisteiden välille muodostaa kolmio (kuva 45).



Kuva 45. Muodostetaan kolmio kärkipisteiden A, B ja C välille.

Kolmiot muodostetaan käyttämällä listan kaikkia pisteitä, jolloin saadaan aikaan kolmioverkko, joka muodostaa vauhtiviivaobjektin geometrian (kuva 46).



Kuva 46. Muodostetaan kolmiot kaikkien pisteiden välille, jolloin muodostuu kolmioverkko, joka voidaan piirtää ruudulle.

UV-koordinaatit generoidaan geometrian luonnin jälkeen. Jokaiselle kärkipisteelle lasketaan UV-koordinaatti jokaisessa animaation vaiheessa.

Ensin luodaan apumuuttuja i_h, jossa i on kyseessä olevan kärkipisteen järjestysnumero:

$$i_h = \frac{i}{2}.$$

Toinen apumuuttuja on i_c, jossa n on kärkipisteiden kokonaismäärä:

$$i_c = \frac{\lceil i_h \rceil}{\frac{n}{2} - 1}.$$

Näistä saadaan U- ja V-koordinaatit. U on siis X-suuntainen komponentti ja V Y-suuntainen. U-komponentti liikkuu tasaisella jaolla arvosta 0 arvoon 1, ja aina kun uusi kärkipiste generoidaan, myös U-komponentti tasataan uudella jaolla. Siksi n on mukana i_c-kaavassa. Se on jaettu kahdella, koska i_h on jaettu kahdella. Toisin sanoen, pisteet tasataan käyttämällä vain puolet kaikista pisteistä, koska toinen puoli pisteistä on samassa kohtaa U-suunnassa:

$$u = 1 - i_c.$$

V-suunnan komponentti on joko 0 tai 1, joten vähennetään järjestysnumerosta järjestysnumeron alaspäin pyöristetty kokonaisluku ja pyöristetään tulos ylöspäin seuraavaan kokonaislukuun:

$$v = \lceil i_h - \lfloor i_h \rfloor \rceil.$$

Vauhtiviivaobjektin generoinnin aloitus määritellään animaatiossa, jossa animaation johonkin kohtaan määritellään tapahtuma (engl. Animation event), joka käynnistää generoinnin ja toinen tapahtuma lopettaa generoinnin. Tämän lisäksi objektille voidaan vielä määritellä, miten pitkään se odottaa generoinnin jälkeen, ennen kuin alkaa hävitä annetussa ajassa pois ruudulta.

5.2 Laavavirtaus

P2-pelissä on muun muassa demoinen susi, jolla on laavaa valuvat silmät. Tehoste on yhdistelmä varjostinohjelmaa ja partikkeleja. Partikkelit luovat silmässä olevat liekit, pienet kipinät ja niistä nousevan savun. Varjostin taas luo virtaavan laavan. Laavavarjostin on muunnelma Valven virtauskartoista ja UV-vierityksestä, ks. luku 4.2.

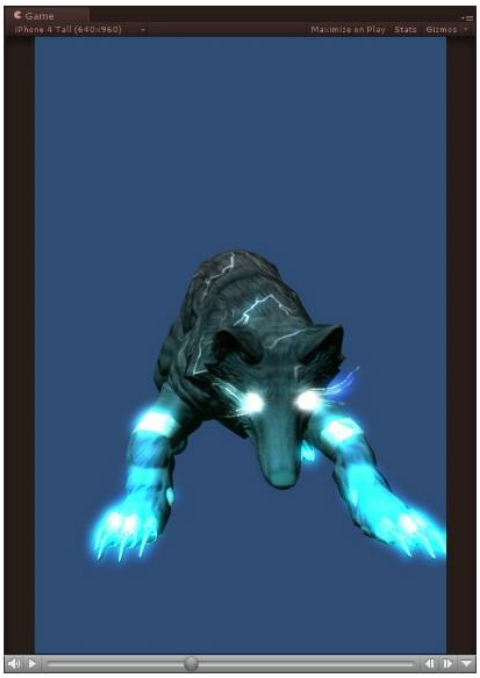
Koska tehoste, kuvassa 47, ei tarvinnut kovinkaan erikoista virtauksen hallintaa, siinä käytettiin virtaukseen UV-vieritystä. Oli kuitenkin tarpeen rikkoa hiukan perinteisen UV-vierityksen lineaarisuutta lisäämällä varjostimeen tekstuurin modulointia venyttämällä sitä eri suuntiin.

Kuva 47. Demoninen susi P2-pelistä.

Tekstuurin venyttäminen saadaan aikaan käyttämällä samaa tekniikkaa kuin Valve, mutta jättämällä pois aikakomponentti. UV-vieritys on tehty luvussa 4.2 esitellyllä tavalla.

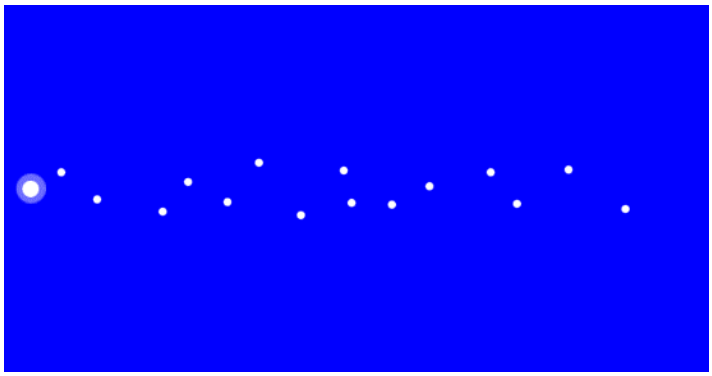
5.3 Partikkeliohjattu transformaatio

Joskus on tarpeen ohjata geometriaa partikkelisimulaatiolla. Tämä toimii tilanteissa, joissa partikkelisimulaatio tuottaa kohtuullisen hallittua liikettä, kuten suunnattu partikkelisuihku. Tässä tapauksessa geometriaa ohjataan partikkeleilla, luomaan heiluva nauhamainen objekti, joka kuvastaa hahmosta tihkuvaa valoenergiaa. Kyseessä on demonisen suden vastakohta valosusi (kuva 48).



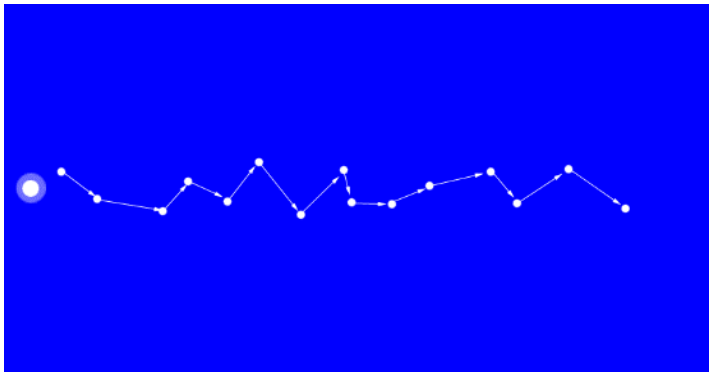
Kuva 48. P2-pelin valosusi.

Itse ohjattava objekti on yksinkertainen levy, jossa on kymmenisen luuta, joilla voidaan taittaa objektia vain muutamasta pisteestä eikä näin tarvitse laskea jokaiselle kärkipisteelle uutta paikkaa koodista vaan luuanimaatio hoitaa väliin jäävät kärkipisteet. Hahmon silmiin on sijoitettu partikkelijärjestelmät, jotka luovat silmistä ulospäin lentäviä partikkeleja. Partikkelijärjestelmissä on myös tuulivoiman tuomaa pientä turbulenssia, ja lisäksi vielä partikkelijärjestelmää käännetään 30 astetta, satunnaisesti eri suuntiin, samalla kun järjestelmä generoi partikkeleja. Tämä luo erikoisenmuotoista partikkelisuihkua, mutta partikkelit jättävät suihkuun ei-toivottuja välejä, eikä niitä voi teksturoida kokonaisuutena (kuva 49). Tästä syystä partikkeleja käytetään ajamaan luita eikä luomaan partikkeleilla näkyvää efektiä.



Kuva 49. Partikkelisuihku.

Levy-objektin viimeisimmän luun tulee seurata partikkelisuihkun vanhinta partikkelia ja seuraavan luun seuraavaksi vanhinta partikkelia ja niin edelleen (kuva 50).

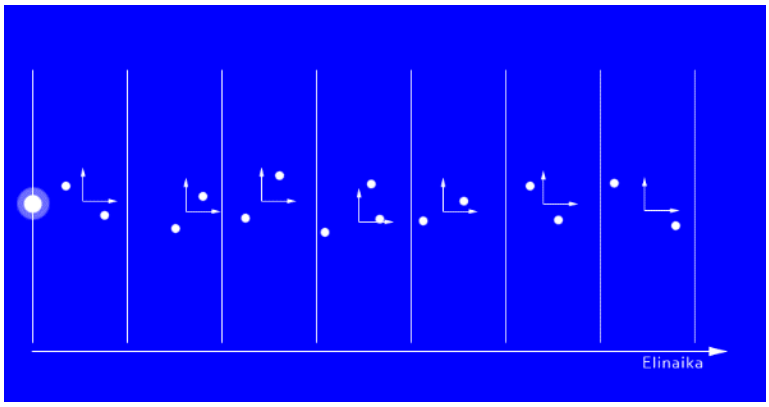


Kuva 50. Nuolet kuvastat partikkelista partikkeliin kulkevaa luuta.

Tämä tekniikka on yksinkertainen tehdä, mutta se vaati, että partikkeleja on järjestelmässä aina saman verran kuin objektissa on luita. Aina ei kuitenkaan ole helppoa varmistaa, että partikkelien määrä on täsmälleen sama. Lisäksi kun käytetään suoraan partikkelin paikkaa, se luo nytkähtelyä aina, kun luu vaihtaa seuraamaan uutta partikkelia. Ongelman voi korjata generoimalla enemmän partikkeleja ja seuraamalla useaa, läheisesti samanikäistä partikkelia samaan aikaan ja ottamalla partikkelijoukosta partikkelien paikkojen keskiarvo.

On tunnettua, kuinka monta luuta objektissa on, joten partikkelit voidaan jakaa yhtä moneen osaan niiden elinaikojen perusteella (kuva 51): vanhin partikkeli järjestelmästä jaettuna luiden määrällä. Kaavassa l on vanhin elinaika ja c on luiden määrä:

$$s = \frac{l}{c-1}.$$



Kuva 51. Partikkelit on jaettu osiin ja L-nuolimuoto osoittaa partikkelijoukon keskipistettä.

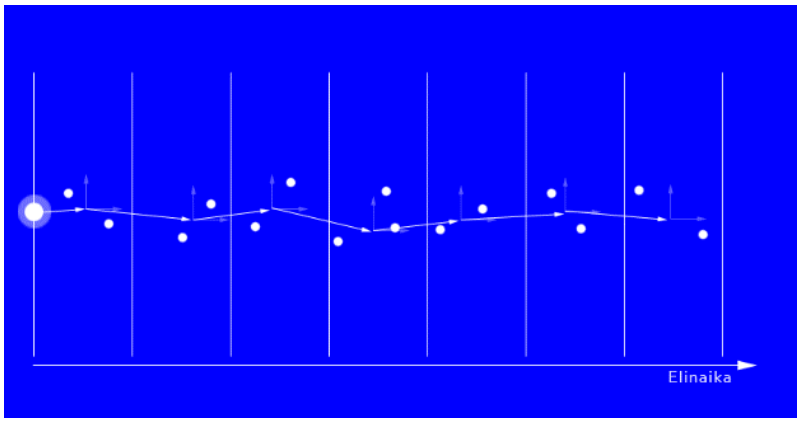
Jokaisessa pelisilmukan kierrossa lasketaan jokaiselle luulle elinaika-alue, jota seurata, pois lukien ensimmäinen luu, joka pysyy paikallaan. Muutoin koko levy-objekti alkaisi liikkua pois paikaltaan partikkelien mukana:

$$ls = s * i$$
$$le = ls + s.$$

Tätä varten pitää tarkistaa, mitkä partikkelit jäävät elinaikojen sisäpuolelle, ja laskea niiden paikoista keskiarvo alla olevalla kaavalla, jossa $\vec{P}$ on partikkelin paikka:

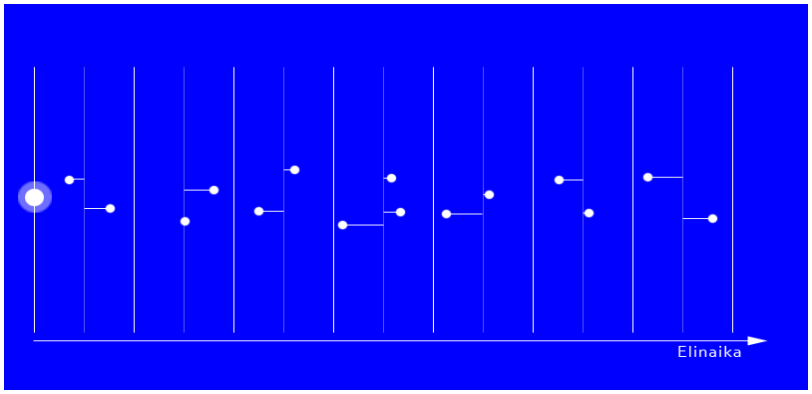
$$pos = \frac{1}{n}\sum_{i=1}^{n} \vec{P_i}.$$

Nyt luut seuraavat useaa partikkelia niiden elinajan perusteella eikä suoraan yhtä partikkelia (kuva 52).



Kuva 52. Nuolet kuvastavat partikkelisuihkulle asettuneita luita.

Tämäkin tekniikka kärsii nytkähtelyistä, koska partikkeli joko vaikutti luuhun tai ei. Ongelman voi korjata seuraamalla enemmän niitä partikkeleja, joiden elinaika on seurattavien elinaikojen keskellä, ja vähemmän niitä, jotka ovat seurattavan alueen reunoilla (kuva 53).



Kuva 53. Kuvassa partikkelista lähtee pieni viiva kohti elinalueen keskustaa. Viiva kuvastaa partikkelin elinajan ja seurattavan elinajan keskustan erotusta. Mitä pidempi viiva sitä vähemmän partikkeli vaikuttaa keskiarvoon.

Ensin selvitetään, millainen vaikutus partikkelilla on eli miten kaukana se on seurattavan alueen keskuksesta. Kaavassa a on elinajan alkuarvo ja b loppuarvo. Kaavalla saadaan elinaika-alueen paikallinen keskipiste, jota tarvitaan vaikutusarvon normalisointiin:

$$m_l = \frac{b - a}{2}.$$

Tarvitaan myös globaali keskipiste:

$$m_g = a + m_l.$$

Nyt voidaan laskea partikkelin vaikutusarvo. l on partikkelin elinaika:

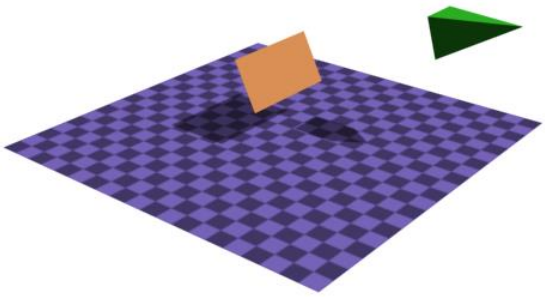
$$w = \frac{m_g - l}{m_l}.$$

Kun tiedetään vaikutusarvo, voidaan laskea painotettu keskiarvo:

$$pos = \frac{\sum_{i=1}^{n} w_i p_i}{\sum_{i=1}^{n} w_i}.$$

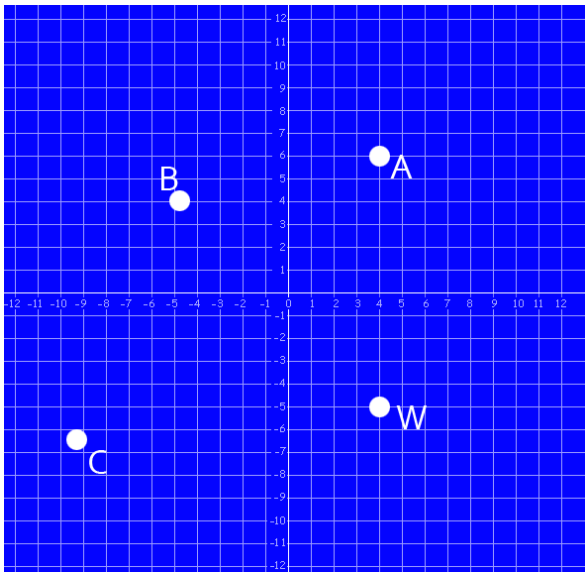
Tämä tapa tuottaa varsin sulavaa animaatiota, kunhan jokaisella alueella on vain muutamia partikkeleita, joista laskea keskiarvo. Liian vähäinen partikkeleiden määrä tuottaa epätarkkuutta animaatioon.

Paikan lisäksi tulee vielä laskea luille niiden orientaatio, jotta luu on kääntynyt siten, että levyn normaali osoittaa kameran suuntaan (kuva 54).



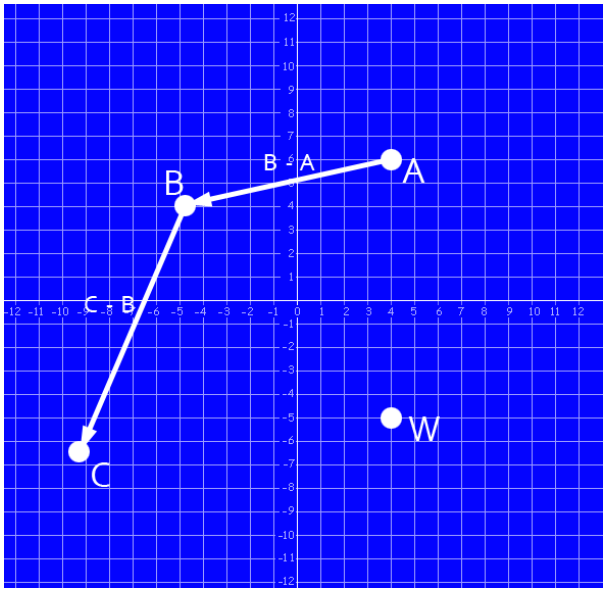
Kuva 54. Luiden piti olla orientoitu siten, että ne katsoivat kohti seuraavaa luuta ja että levyn normaali osoitti kohti kameraa.

Ajatellaan, että on olemassa kolme luuta (A, B ja C) ja kamera (W) (kuva 55).



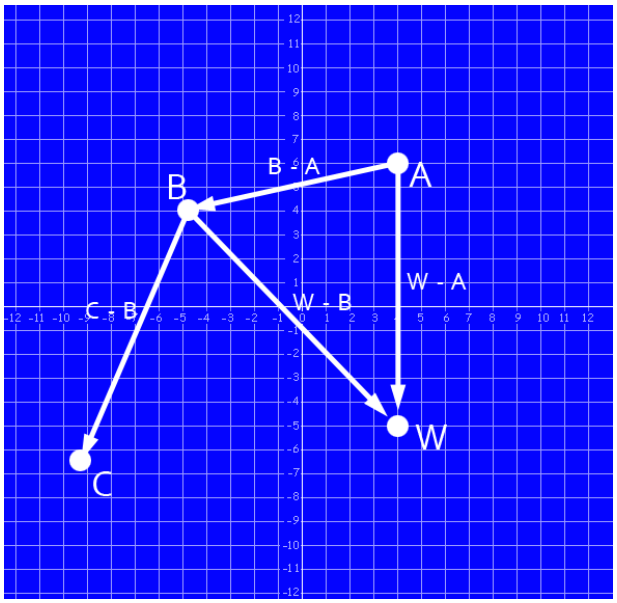
Kuva 55. Kolme luuta ja kamera 2D-tasolla.

Lasketaan suuntavektorit luusta A luuhun B ja B:sta luuhun C (kuva 56).



Kuva 56. BA- ja CA-suuntavektorit.

Luista otetaan suuntavektorit kameraan, paitsi viimeisestä luusta C. C-luu ei ole oikeasti kiinni yhdessäkään objektin kärkipisteessä, vaan se on mukana vain siitä syystä, että saadaan laskettua luun B orientaatio (kuva 57).



Kuva 57. Suuntavektorit luista A ja B kameraan W.

Nyt voidaan laskea luun Y-akselin kiertovektori ottamalla ristitulo kahdesta jo lasketusta vektorista:

$$\vec{Y} = \vec{Z} \times \vec{X}.$$

Näin saadaan aikaan vektori, joka on kohtisuorassa vektoreita Z ja X vastaan. X-akseli saattaa kuitenkin olla tässä vaiheessa vinossa eli ei 90 asteen kulmassa Z-akselia vastaan, joten X-akselin suunta pitää vielä varmistaa laskemalla ristitulo:

$$\vec{X} = \vec{Z} \times \vec{Y}.$$

Nyt jokaiselle luulle voidaan luoda orientaatiomatriisi R:

$$R = \begin{matrix} X.x & X.y & X.z \\ Y.x & Y.y & Y.z \\ Z.x & Z.y & Z.z \end{matrix}$$

Lopuksi matriisista vielä konvertoidaan quaterniot ja sijoitetaan ne luulle. Tähän voidaan käyttää Unity API:n funktiota Quaternion.LookRotation.

5.4 Muut tehosteet

P2-pelin ensimmäiseen esittelyversioon tehtiin useita tehosteita, joista kaikissa ei ollut teknisesti mitään erikoista tai niitä varten ei kehitetty uutta omaa teknologiaa, mistä syystä näitä tehosteita ei yksityiskohtaisesti käyty läpi tässä työssä. Tässä luvussa kuitenkin esitellään muutamia tehosteita ilman teknistä selostusta niiden toiminnasta.

Vauhtiviivatehostetta käytettiin muun muassa lyömäaseen tehosteena, kuten näkyy kuvasta 58.

Kuva 58. Vauhtiviivatehoste. Oikea puoli on pelinäkymästä ja vasemmalla sama tehoste vähän paremmasta kulmasta työtilanäkymässä.

Vauhtiviivatehosteesta on tarkempi kuvaus luvussa 5.1.

Insinöörityön osana tutkittiin myös kustannustehokkaan kasvoanimaation mahdollisuuksia ja kasvojen graafisen tason mahdollisuuksia iOS-mobiililaitteella. Lopputulos näkyy kuvassa 59.

Kuva 59. Kasvotesti iOS-laitteella.

Pelin hahmoilla on myös ampuma-aseita, jotka tarvitsivat liekkitehosteet ja luotien valojuovat, jotka ovat nähtävissä kuvassa 60.

Kuva 60. Aseen liekkitehoste ja luodin valojuova.

Luodin valojuova käyttää samaa vauhtiviivatehostetta kuin lyömäaseetkin. Ainut ero on erilainen tekstuurikuva, mutta se jakaa saman ohjelmakoodin. Liekkitehoste on toteutettu Unity-moottorin Shuriken-hiukkassimulaatiolla.

Pelaajalle on jotenkin kerrottava, että hän aiheutti vahinkoa tai vastaanotti vahinkoa. Kuvassa 61 näkyy, miltä näyttää, kun pelaaja itse saa vahinkoa.

Kuva 61. Pelaaja saa vahinkoa.

Tehosteen teksti kertoo, kuinka paljon suden purema aiheutti vahinkoa. Reunoilla oleva punainen reunus indikoi, että pelaaja oli vahingon kohde, ja ruudulla näkyvä keltainen tähti kertoo, mihin kohtaan ja mihin hahmoon vahinko kohdistui.

Pelaaja voi myös parantaa hahmojaan, ja sitä visualisoitiin kuvan 62 esittämällä tavalla.

Kuva 62. Parannustehoste.

Tehoste on tehty Unity-moottorin hiukkassimulaatiolla.

Esiteltyjen tehosteiden lisäksi tehtiin myös joukko muita visuaalisia tehokeinoja, jotka eivät kuitenkaan olleet osa tehostejärjestelmää. Niitä olivat ruudun himmentyminen ja ruudun reunojen himmentäminen, kameratehosteita, kuten kuvakulmien vaihtoja ja hidastuksia, sekä muutamia varjostimia, joista mielenkiintoisimmat esiteltiin tässä työssä.

6 Yhteenveto

Insinöörityössä tutkittiin tietokonepelien tehosteiden historiaa, ja huomattiin, että joitain tehostetekniikoita, jotka on kehitetty jo 35 vuotta sitten, on edelleen käytössä. Tehosteiden kehityksen yhteydessä tutkittiin laajasti muiden valmistajien tekemiä tehosteita. Referenssejä haettiin useilta alustoilta ja eri tyyleistä, joiden avulla tässä työssä esitellyt tehosteet ovat kehittyneet siihen muotoon missä ne nyt ovat.

Tehosteiden visuaalisuuden lisäksi tutkittiin myös tehosteiden teknistä toteutusta ja algoritmeja, joita niitä varten on kehitetty. Monien työssä esiteltyjen tehosteiden toteutustapa perustuu johonkin olemassa olevaan tekniikkaan tai algoritmiin. Varsin merkittävä osa kehitysajasta kului erilaisten tutkimusraporttien lukemiseen ja seminaarien katsomiseen. Kehittäjätiimi kävi kehitystyön aikana Seattlessa asti, Unite 2014 -konferenssissa, saamassa oppia uusimmista tekniikoista.

Tehosteita kehittäessä esiin tuli monta seikkaa, jotka eivät etukäteen olleet tiedossa. Peliä kehitetään Unity-moottorilla, joka ei ollut entuudestaan kovinkaan tuttu, joten sen käyttämässä API:ssa oli toisinaan yllättäviä teknisiä toteutuksia. Unityn käyttämässä C#-kielessä oli myös yllättäviä toteutustapoja liittyen lähinnä muistinhallintaan (ks. luku 4.6). Itse tehosteiden luonnin lisäksi piti myös kehittää työkaluja niiden tekemiseen ja datan tallentamiseen sopivaan muotoon. P2-pelin kehityksessä käytetyt ohjelmistot olivat aikaisemmasta kokemuksesta osin poikkeavia, joka alkuun aiheutti epäselvyyttä. Useita insinöörityön aikana kehitettyjä tehosteita myös hylättiin ja tehosteen suunnitelmaa jouduttiin parantamaan, jotta tehoste viesti pelaajalle haluttua asiaa.

Työn lopputuloksena syntyivät tehosteet P2-pelin ensimmäiseen esittelyversioon, jonka perusteella yrityksen johto päättää, saako peli jatkorahoitusta. Jatkorahoitus saatiin demon esittelyn jälkeen, joten projekti oli onnistunut. Esittelyversion kehityksen aikana tiimissä oli kuusi jäsentä, ja kehitys aloitetiin tyhjästä. Tästä syystä peli graafinen ilme on vielä varsin karu, mutta tarkoitus oli todistaa, että peli on mahdollista ja tehdä ja että se voi olla hyvä. Nykyisin pelin parissa työskentelee noin 20 kehittäjää ja peliä työstetään kohti lopullista julkaisua. Tehosteita edelleen jatkokehitetään ja vaihdetaan uusiin aina sen mukaan, miten pelin graafinen ilme elää.

Lähteet

Alhman, Sebastian. 2014. Johtava ohjelmoija, Remedy Entertainment, Helsinki. Haastattelu. 4.11.2014.

Pirinen, Tuomas. 2015. Johtava ohjelmoija, Remedy Entertainment, Helsinki. Haastattelu. 30.1.2015.

Beam, Josh. 2009. Introduction to Software-based Rendering: Triangle Rasterization. Verkkodokumentti <http://joshbeam.com/articles/triangle_rasterization>. 19.1.2009. Luettu 6.2.2015.

Mass Effect 2. 2014. Verkkodokumentti. Bio Ware. <http://masseffect.bioware.com/me2/>. Luettu 5.2.2015.

Compression of JavaScript Programs. 2011. Verkkodokumentti. Bits'n'Bites. <http://www.bitsnbites.eu/?p=20#better_png>. 24.4.2011. Luettu 6.2.2015.

Textures And Patterns. Verkkodokumentti. CZE. <http://mirror2.cze.cz/textures-Large/water-texture.jpg>. Luettu 6.2.2015.

Gronow, Nick. 2012. Tail Arc Renderer. Verkkodokumentti. <http://wiki.unity3d.com/index.php?title=Trail_Arc_Renderer>. 10.1.2012. Luettu 6.2.2015.

Hastings, Dan. 2014. The History of Wolfenstein Games. Verkkodokumentti. Nerd Burglars. <http://nerdburglars.net/263/Article/the-history-of-wolfenstein-games.html>. 17.5.2014. Luettu 6.2.2015.

Lammers, Kenny. 2013. Unity Shaders and Effects Cookbook. Packt Publishing.

Lamothe, Andre. 1999. Tricks of the Windows Game Programming Gurus. SAMS.

Monkey Island. 1990. Verkkodokumentti. Lucas Arts. <http://retrodungeon.net/?p=125>. Luettu 20.9.2014.

MSDN: int (C# Reference). 2013. Verkkodokumentti. Microsoft. <https://msdn.microsoft.com/en-us/library/5kzh1b5w.aspx>. Luettu 6.2.2015.

Spacewar!. 2005. Verkkodokumentti. Computer History Museum. <http://pdp-1.computerhistory.org/pdp-1/?f=theme&s=4&ss=3>. 1.3.2005. Luettu 6.2.2015.

Pac-Man. 2014. Verkkodokumentti. Namco Bandai Games. <http://pac-man.com/en/pac-man-history/>. Luettu 6.2.2015.

Realm Of Gaming - Heavenly Sword, Screenshots. 2007. Verkkodokumentti. Ninja Theory. <http://www.realmofgaming.com/screenshots/playstation3/heavenly-sword/928391_20070711_screen003.jpg>. Luettu 6.2.2015.

Explosion Generator. Verkkodokumentti. Positech Games. <http://www.positech.co.uk/content/explosion/explosiongenerator.html>. Luettu 6.2.2015.

Remedy Games. 2014. Verkkodokumentti. Remedy Entertainment Ltd. <http://remedygames.com >. Luettu 6.2.2015.

Schuller, Daniel. 2011. C# Game Programming: For Serious Game Creation. Course Technology.

Visual Walkthrough. 2009. Verkkodokumentti. tomb-raider-anniversary.com. <http://tomb-raider-anniversary.com/tomb-raider-anniversary-walkthrough/Level14_Final_Conflict/index3.php>. Luettu 6.2.2015.

Unity Manual - Particle System. Verkkodokumentti. Unity. <http://docs.unity3d.com/Manual/class-ParticleSystem.html>. Luettu 6.2.2015.

Unity Manual - Shader Reference. Verkkodokumentti. Unity. <http://docs.unity3d.com/Manual/SL-Reference.html>. Luettu 6.2.2015.

Unity Manual - Tail Renderer. Verkkodokumentti. Unity. <http://docs.unity3d.com/Manual/class-TrailRenderer.html>. Luettu 6.2.2015.

Winter, David. 2013. Pong Story. Verkkodokumentti. <http://www.pong-story.com/odyssey.htm>. Luettu 6.2.2015.

Vlachos, Alex. 2010. Water Flow in Portal 2. Verkkodokumentti. Valve Software. <http://www.valvesoftware.com/publications/2010/siggraph2010_vlachos_waterflow.pdf>. Luettu 6.2.2015.

Wloka, Matthias. 2009. "Batch, Batch, Batch:" What Does It Really Mean. Verkkodokumentti. nVidia. <http://www.nvidia.com/docs/IO/8228/BatchBatchBatch.pdf>. Luettu 6.2.2015.

Max Payne. 2002. Verkkodokumentti. Rockstar Games. <http://www.rockstargames.com/maxpayne/main.html>. Luettu 4.3.2015.

Street Fighter 4 – Official Trailer. 2009 Verkkodokumentti. Capcom. <https://www.youtube.com/watch?v=2uOt-XmNrQw>. Luettu 5.3.2015.

Tekken 6 Gameplay. 2010. Verkkodokumentti. Namco Bandai Games. <https://www.youtube.com/watch?v=X55AWQjEaQ8>. Luettu 5.3.2015.

Final Fantasy 13 Battle Gameplay. 2010. Verkkodokumentti. Square Enix. <https://www.youtube.com/watch?v=GMdDEodrNQI>. Luettu 5.3.2015.

Corefire Imp – Heal. Verkkodokumentti. Blizzard. <http://gamediplomat.com/wp-content/uploads/2013/02/CorefireImp_Heal-770x285.jpg>. Luettu 5.3.2015.

Healing Wave. Verkkodokumentti. Square Enix. <http://finalfantasy.wikia.com/wiki/Healing_Wind_(Ability)?file=VIICC_Healing_Wave.jpg>. Luettu 5.3.2015.

Ninja Gaiden 3. 2012. Verkkodokumentti. Team Ninja. <https://www.youtube.com/watch?v=Q162oZJoObk>. Luettu 5.3.2015.

Zelda – Hyrule Warriors. 2014. Verkkodokumentti. Koei Temco. <https://www.youtube.com/watch?v=UX_NCD0ey6o>. Luettu 5.3.2015.

Kritika. 2014. Verkkodokumentti. Gamevil. <https://www.youtube.com/watch?v=7FeUsuvt8wE>. Luettu 5.3.2015.

Lava Turtle. Verkkodokumentti. Blizzard. <http://i864.photobucket.com/albums/ab207/cataclysmexplorers/General/Terrorpene_Shindos_1.jpg>. Luettu 5.3.2015.

Tomb Raider Anniversary – Level 14. Verkkodokumentti. Crystal Dynamics. <http://media.tools4noobs.com/Level14_Final_Conflict/Level14_Final_Conflict_image024.jpg>. Luettu 5.3.2015.

Oblivion Gate. Verkkodokumentti. Bethesda Software.<http://img3.wikia.nocookie.net/__cb20120103090750/elderscrolls/images/1/15/Oblivion-Gate.jpg>. Luettu 5.3.2015.

Portal 2. Verkkodokumentti. Valve Software. <http://www.cosplayisland.co.uk/files/costumes/855/41850/Chell_p2_thruportal.jpg>. Luettu 5.3.2015.

X-Com Enemy Unknown VS. Aftermatch. Verkkodokumentti. Mythos Games. <http://s4.photobucket.com/user/Uberbucket/media/UFO%20Lets%20Play/46%20June%2017-18/10FourthAlienDead.png.html>. Luettu 5.3.2015.

They Play – Grimrock. 2014. Verkkodokumentti. Almost Human. <https://www.youtube.com/watch?v=Xp9S2R-utZU>. Luettu 5.3.2015.

Haastattelu Tuomas Pirinen

K: Kuka olet ja mitä teet työksesi?

V: Tuomas Pirinen, töissä Remedy Entertainment:lla nimikkeellä Head of Game Design.

K: Mikä on roolisi P2 pelissä?

V: Olen vastuussa kahdesta asiasta. Toinen on pelimekaniikan suunnittelu ja toinen vision hallinta eli ison linjan projektinjohto. Yritän tässä pitää hyvin vapaata otetta, eikä aasialainen tai pohjoisamerikkalainen ”fuhrer-henkien” johtaminen sovi minulle. Toivon ennemmin tiimin jäsenten oman näkemyksen tulevan esille, mutta jos asia sitä vaatii niin, silloin olen valmis näyttämään suuntimaa.

K: Mitä olet tehnyt aikaisemmin?

V: Se onkin pitkä tarina. Tiivistetään 20 vuotta alla. Gamershop:lla, Englannin Notthinghamssa, olin vastuussa Warhammer Fantasy Battle:n kuudennesta editiosta, eli tein lautastrategiapelejä (eng. table top strategy). Aloitin Morthaimer IP:n (intellectual property, suom. immateriaalioikeus) ihan alusta alkaen. Tein myös suuren määrän maailman luontia, useita armeijalistoja, työskentelin taiteilijoiden, veistäjien ja keskustelin business ihmisten kanssa yrityksen strategista, mutta ennen kaikkea kirjoitin sääntöjä peleihin. Sen jälkeen työskentelin Eidos:lle useankin pelin kanssa, pari vuotta. Siitä Microsoftille pariksi vuodeksi, jossan tein Surek peliä. Sitten EA:lle, työskentelin lukuisissa EA Canadan peleissä, joista varmaan merkittävin oli Need for Speed sarja. Seuraavana Ubisoft, jossa tein mm. Driver peliä. Olen tehnyt pelejä lautapeleista, mobiilista, konsoliin. Olen aina ollut peliagnostikko eli minulle ei ole tärkeää millä alustalla peli pyörii vaan onko se hyvä vai ei.

K: Miksi P2?

V: Paljon riippuu siitä millainen peli studiolle sopii ja Remedy on hahmovetoinen studio. En edes halunnut alkaa tekemään peliä, joka ei sovi studiolle, jos esim. pistäisi Grand Turismo:n tekijöiden tekemään seuraavaa Call of Duty:a niin siitä tuskin tulisi kovin hyvää. Projektilla oli parametreina, että sen pitää olla Remedy-henkinen peli, mutta sillä

täytyy olla mahdollisuus suurmenestykseen, jonka vuoksi peli on loputon, joka on nykyisin osoittautunut hyvin menestyksekkääksi. Tarkoitus on yhdistää Remedy:n luova ja taiteellinen näkemys menestyskonseptiin ja tämän henkiset pelit ovat lähellä sydäntäni.

K: Sinä olet ollut mukana, useassakin pelissä, ihan alusta ihan loppuun. Millainen on pelin suunnitteluprosessi?

V: Tärkeä asia on päättää ihan alussa, että minkälaista peliä tehdään. Sellaista maagista luovuutta, että nyt keksitään ihan kaikki tyhjästä ei juurikaan ole. Kannattaa katsoa mikä on genren ja tyylin kilpailutilanne ja sitten miettiä millä me voidaan innovoida. Jos tehdään first person shooter peliä niin tarvittaisiin oikein todella hyvä syy miksi ei laitettaisi ampumisnappia oikeaan shoulder button:iin. Toki jos keksitään jotain todella hyvää millä peli erottuu edukseen muista niin se kannattaa tehdä, mutta siitä ei kannata lähteä liikkeelle. Luovuuden ja genren ymmärtämisen yhteistyö on minusta todella tärkeää. Tärkeää on myös tarina, taidesuunta ja muu, mutta ensin pitää olla rehellinen sen suhteen millaista peliä ollaan tekemässä. Tehdään sitten first person cover shooter, mutta tehdään siitä ainakin genrensä paras.

K: Lähteekö sinusta pelisuunnittelu tarinasta vai ideasta?

V: Riippuu pelistä, jos tehdään kilpailijaa Heavy Rain pelille niin parempi lähteä liikkeelle tarinasta. Kilpailijaa Street Figther:lle niin turha lähteä tarinasta liikkeelle, jos ei pärjää Capcom:lle pelimekaniikassa.

K: Voiko alkaa tekemään peliä ajatuksella ”Halutaan peli X, koska sellainen myy”?

V: Voi. EA:han teki Fifa:n. Ne näki paljonko Pro Evolution Soccer myi ja sanoivat, että tämä markkina vallataan. Ostivat Fifa:n lisenssin ja parhaat jalkapallopelien tekijät, iteroivat kunnes nykyisin joka vuosi tulee 90 Meta Critic pisteen peli. Tämä ei missään tapauksessa toimi, joka genressä. Riippuu paljon siitä mille tontille ollaan menossa.

K: Kun alat suunnittelemaan peliä niin otatko huomioon moraalisia seikkoja, kuten onko pelissä seksismiä, rasismia tai muuta halventavaa sisältöä.

V: Kyllä. Siihen on sekä eettisiä, että business syitä. Eettiset syyt on ihan yksinkertaisia, jonka voi ottaa vaikka elämän ohjesäännöksi ”Don’t be an asshole”. Ei tarvitse tehdä tuotteita, jossa pelaaja kannustetaan ikävään sosiaaliseen käyttäytymiseen. En kuitenkaan ole mikään moraalinkukkanen olihan Need For Speed:kin kaahaamista. Siihen tosin lisättiin varoitusteksti pelin alkuun, että kyse on fiktiosta eikä tätä pidä tehdä oikeassa elämässä. Business näkökulmasta esim. naistenhalventamisella menettää asiakkaita toki jotkut pelit taas käyttävät sitä saadakseen asiakkaita. Riippuu kenelle peliä tehdään ja mitä halutaan pelillä sanoa. Se ei ole se tapa jolla minä haluan mennä. P2:ssa aivan tarkoituksen mukaisesti valittiin toiseksi päähahmoksi musta nainen. Ei niinkään siksi, että haluaisin ottaa kantaa minkään asian puolesta vaan halusin lähinnä mielenkiintoisen hahmon.

K: Onko alustan haltijalla sanomista siihen miten paljon raakuutta pelissä saa olla?

V: On, mutta usein salittaan raa’at pelit, kunhan niiden ikärajat ovat asianmukaiset. En myöskään ole suuri sensuurin ystävä. Voin kenties olla pitämättä jostain pelistä sen maailmankuvan takia, mutta en alkaisi kieltämään sitä. Seksin ja väkivallan kanssa on kuitenkin enemmän tekemistä eri maiden lainsäädännön kanssa, kuin alustan. Esimerkiksi Saksassa natsi-ihannointi on laitonta. Sitten ihan porno on yleensä kielletty jo alustanhaltijankin puolesta. En kenties aina tajua miksi seksi on niin paljon pahempaa, kuin päiden katkominen, mutta näin se nyt vain meidän kulttuurissa on.

K: Uskotko siihen, että konsolit kuolee?

V: En. Niiden käyttäjämäärät eivät ehkä aivan ole iOS tasolla, mutta kyllä jo pelkästään PS4 pääsee tänä vuonna sinne 14-16 miljoonan kappaleen myyntiin. Ongelma on pelin kehityskustannukset. Ennen peliä saatettiin myydä puolimiljoonaa kappaletta, kaikki sai bonuksen ja palkat voitiin korottaa ja kaikki lähti iloisena seuraavaan projektiin. Nykyään se olisi katastrofi. Pelaajakunnan pitäisi kasvaa samassa suhteessa kehityskustannuksien kanssa ja näin ei ole tapahtunut. Toinen on sitten Mooren laki, että meillä on kohta niin kovaa rautaa joka paikassa tv:ssä, kännykässä, tabletissa ettei tarvita mitään muuta. Konsolipelien kehityskustannusongelma johtaa ikävästi siihen, että niiden genrevalikoima harvenee. Vain menestyneet pelit jää eloon Call of Duty, Skyrim, Fifa ja eikä paljon muista. Se on ikävää, koska olen aina pitänyt siitä, että on erilaisia pelejä. Ymmärrän toki julkaisijaakin, jos peli tuottaa tappiota niin sitä ei tehdä.

K: Millainen on P2 pelin aloitustilanne?

V: Pelaaja on auto-onnettomuudessa ja vaikuttaa siltä, että kuolet, avaa silmäsi ja huomaa olevansa Yhdysvalloissa, mutta korruptoituneessa versiossa. Suuriosa ihmisistä on kadonnut ja maailma on selvästi jonkinlaisen pahan vallassa. Taskussasi on kirje tyttäreltäsi. ”Olen San Francossa, tule apuun äiti/isä”. Pelaaja voi pelata joko mies -tai naishahmoa. Kaikki ihmiset eivät ole kadonneet ja muut ovat kutakuinkin samassa tilanteessa, kuin sinä. Niinpä lyöttäydytte yhteen, otatte auton ja lähdette ajamaan Amerikan halki. Matkalla etsitään ruokaa, polttoainetta ja taistellet meidän maailmastamme heijastuvia pahoja tapahtumia vastaan, jotka tässä maailmassa ottavat hirviöiden muodon.

K: Pelissä mennään läpi Amerikan, miten se tapahtuu?

V: Pelissä on kaksi karttaa. Toisessa näkyy koko Yhdysvaltojen kartta ja edistymisesi siinä. Toisessa kartassa näkyy lähialue. Lähialueen kartassa näkyy rakennukset ja muut yksityiskohdat. Tarvikkeet on pelissä kokoajan loppu ja niitä pitää etsiä paikoista matkan varrella.

K: Miten se tapahtuu?

V: Ensin perustat leirin. Sitten valitaan kolmen hengen joukko eli kaksi, kaikista mukana olevista ihmisistä, pelaajan lisäksi, jotka alkavat tutkia ympäristöä. Ajatuksena on, ettei vaaranneta koko ryhmää ja jätetään muut vartioimaan leiriä ja keräämään tarvikkeita. Kolmen hengen ryhmän kanssa taistellaan hirviöitä vastaan, joita tässä kyseissä paikassa on.

K: Jos taistelussa joku kuolee niin mitä tapahtuu?

V: Käytämme termiä Knock Out, kun hahmo ”kuolee”. Pelaaja on voinut käyttää paljon vaivaa tai jopa oikeaa rahaa saadakseen hahmoja, jolloin niiden kokonaan pois ottaminen olisi kohtuutonta.

K: Tuleeko siitä kuitenkin jokin sanktio?

V: Kyllä. Hahmot eivät kerää kokemusta ja taisteluteho heikkenee. Aiomme myös kokeilla tapaa, että hahmo joutuisi toviksi sivuun kunnes on taas parantunut.

K: Montako hahmoa voi olla kerralla mukana?

V: Se riippuu autosta, joka määrää montako siihen mahtuu. Hahmoja saa mukaansa lisää hankkimalla isomman auton.

K: Miten taistelu toimii?

V: Taistelu on ns. Puzzle Combat. Eli ruudulla on ikoneja, jotka vastaavat hahmojen toimintoja. Raahaamalla ruudulle ampumaaseikonin, hahmot joilla on ase ampuvat ja raahaamalla ruudulle puukkoikonin hahmot käyttävät mahdollista lähitaisteluominaisuuksiaan. Ikoneiden kanssa voidaan muodostaa erilaisia yhdistelmiä ja kuvioita, jotka antavat hard-core pelaajille haastetta ja syvyyttä.

K: Miten hahmokehitys toimii?

V: Hahmot taistellessaan keräävät kokemuspisteitä ja nousevat tasoja. Vihollisilta löytyy myös tarotkortteja. Joilla voidaan entisestään buffata hahmojen taitoja. Korteilla voidaan myös muodostaa yhdistelmiä, jotka ovat voimakkaampia, kuin yksittäinen kortti.

Haastattelu Sebastian Ahlman

K: Kerro kuka olet ja mitä teet?

V: Olen Sebastian Alhman, johtava ohjelmoija. Minut on alun perin palkattu peliohjelmoijaksi, joskin olen kuitenkin tehnyt pääosin teknologiaohjelmointia. Nyt johdan P2 projektin ohjelmointitiimiä.

K: Mitä olet tehnyt aikaisemmin?

V: Olen aikaisemmin ollut töissä Bugbearilla ja Digital Chocolatella. DC:llä melkein pari vuotta ja Bugbearilla runsaan vuoden ja nyt Remedyllä kaksi ja puolivuotta. Ennen pelialalle tuloa tein viitisen vuotta hyötyohjelmistokehitystä. Eli kymmenisen vuotta olen ollut ohjelmointityössä. Toki harrastepohjalta olen aina tehnyt pelejä.

K: Tehdessä peliä mobiilialustalle niin mitä asioita tulee ottaa erityisesti huomioon?

V: Vähemmän muistia ja hitaampi CPU, jos verrataan PC:hen.

K: Jos peli ei pyöri, mitä teet ensimmäisenä?

V: Ensin profiloidaan eli selvitetään mistä on kyse. Jos peli yskii eli pyörii muutoin hyvin, mutta joissain kohdin jämähtää toviksi paikalleen niin on syy voi olla ajonaikaisessa muistin varauksessa. Yleensä, ainakin meidän projektissa, on grafiikkapuoli eli raskaat materiaalit ja varjostimet.

K: Kerro mitä on pino –ja kekomuisti?

V: Keko on dynaamista muistia, josta tietokone joutuu etsimään vapaan muistialueen ja pitämään siitä kirjaa. Pino on muistia, joka on varattu funktiolle ja käytössä kyseisen funktion sisällä.

K: Miksi pinomuisti on kekoa nopeampaa?

V: Pinomuisti on valmiiksi varattua muistia, jonka osoitteet tiedetään jo valmiiksi ja on sen vuoksi tietokoneelle yksinkertaisempaa käyttää. Miten keko –tai pinomuisti lopulta toimii on kiinni kääntäjän toteutuksesta. Kyse on samasta muistista, jota vain käsitellään eri tavoilla.

K: Miten varmistut siitä, ettei muisti koskaan lopu kesken?

V: Budjetoidaan muisti, joka tarkoittaa sitä, että jokaiselle systeemille on olemassa oma budjettinsa. Quantum Breakissa minulla oli 200 megaa state recorder järjestelmää varten, joka varattiin sille. Sitten state recorder sisäisesti käytti sitä muistia, joka sille oli varattu. Sama tehdään jokaiselle järjestelmälle ja näin pidetään voidaan olla varmoja ettei muistia käytetä liikaa.

K: Entä jos muisti kuitenkin loppuu kesken? Mitä silloin tapahtuu?

V: Riippuu paljon pelinmoottorista. Joissain moottoreissa saattaa olla jokin virheenkäsittelijä tätä varten, mutta se on hyvin hankalaa ja usein sellaista ei ole. Yleinen tilanne on, että ohjelma antaa ”OutOfMemory” virheen ja käyttöjärjestelmä sulkee ohjelman.

K: C# hoitaa muistinhallinnan automaattisesti. Tarkoittaako se, ettei muistista tarvitse enää murehtia?

V: Ei tarvitse vapauttaa muistia itse, mutta kyllä siitä silti pitää murehtia. Jotkut Unityn funktiot esim. CreateTexture varaa natiiviamuistia (eli C# automaattisenmuistinhallinnan ohi), joka pitää itse vapauttaa. Tietysti pitää olla tarkkana myös siitä, ettei varaa kekomuistia ajonaikana.

K: Jos peli on vie liikaa muistia. Mistä säästät ensin?

V: Profiloidaan ensin. Yleensä tekstuurit ja 3d-objektit tarkistetaan ensin, ettei niistä ole GPU ja CPU kopioita. Usein riittää, että tekstuuri ja 3d-objekti on vain GPU:n muistissa. Tämän voi Unityssa säätää Read/Write flagilla.

K: Entä jos peli on liian suuri? Mistä säästät ensin tilaa?

V: Riippuu pelistä. Tekstuurit kuitenkin on hyvä paikka aloittaa. Videot kannattaa tarkistaa, että voiko ne pakata paremmin. Joskus tiedoston kirjoittaminen binäärinä voi auttaa paljonkin tiedoston koossa verrattuna tekstimuotoiseen tiedostoon. Joskus kuitenkin tekstimuotoinen tiedosto voi olla pienempikin. Riippuu paljon siitä mitä dataa sinne tallennettaan.

K: Onko pelin koolla ja muistin kulutuksella yhteyttä? Tarkoittaako isompi peli aina isompaa muistin kulutusta?

V: Yleensä kyllä, jos peli on iso se usein tarkoittaa paljon assetteja, jotka pitää ladata muistiin. Ei se kuitenkaan välttämättä sitäkään tarkoita. Voihan tehdä vaikka 1K demon, joka generoi paljon dataa vaikka pelin koko levyllä ei olekaan iso.

K: Suositteleko säikeiden käyttöä Unityssa?

V: Unityssa voi tehdä ihan tavallisia .NET säikeitä, mutta puhelinlaitteella ei välttämättä hirveästi edes säikeistä saa etua sillä monessakaan laitteessa ei vielä ole multicore prosessoreja. Lisäksi tulee osata kirjoittaa multicore-koodia, jotta koodista tulee tehokasta. Unity mainostaa Coroutine systeemiään kevyinäsäikeinä, vaikkei niissä varsinaisesti ole kyse säikeistä. Joskin jonkin verran ulkoisesti näyttävätkin niiltä.

K: Miten tärkeää on tuntea käytetty alusta, jotta sille voi kirjoittaa optimaallista koodia?

V: Todella tärkeää. Harvoin tosin tuntee, ellei keskity yhteen alustaan pitkäksi aikaa.

K: Milloin jokin asia kannattaa tehdä CPU:lla ja milloin GPU:lla?

V: GPU on hyvä tekemään rinnakkaislaskentaa. Eli teet jonkin yhden yksinkertaisen asian, mutta teen sen miljoona kertaa. Esim. meidän projektissamme pystyimme analysoidaan oclusion datat parissa sekunnissa, joka vei CPU:lla useamman minuutin.

K: Koodi jota kirjotetaan Unityyn ei päädy sellaisenaan peliin, vai päätyykö?

V: Riippuu mihin alustaan käännetään. iOS alustella koodi muunnetaan C++ koodiksi ja esikäännetään, mikä tarkoittaa sitä, että kaikki C# ominaisuudet ei toimi iOS:lla. Esim. LINQ kyselyt eivät toimi, koska ne vaativat ajonaikaisen koodigeneroinnin.

K: Milloin kannattaa käyttää rakenteita eikä luokkia?

V: Jos haluat syöttää datatyypin jonnekin viittauksena niin on pakko käyttää luokkia, mutta jos haluaa ne arvoina niin pitää käyttää rakennetta.

K: Mitä tarkoittaa memory pooling?

V: Muisti on varattu valmiiksi ja samaa muistia uudelleen käytetään. Eli muistin varaamista ei tapahdu uudestaan missään vaiheessa.