Teemu Heikkinen, Pavel Kauhanen, Antti Pikkarainen

XNA Pelikehitysympäristönä

XNA Pelikehitysympäristönä
Kajaanin ammattikorkeakoulu
Tradenomikoulutus
Tietojenkäsittely
Syksy 2008

Kajaanin ammattikorkeakoulu

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Teemu Heikkinen, Pavel Kauhanen, Antti Pikkarainen	
Työn nimi XNA Pelikehitysympäristönä	
Vaihtoehtoiset ammattiopinnot Ohjelmistosuunnittelu	Ohjaaja(t) Veli-Pekka Piirainen
	Toimeksiantaja Veli-Pekka Piirainen / Kajaanin Ammattikorkeakoulu
Aika 18.11.2008	Sivumäärä ja liitteet 102 + 1

Opinnäytetyössä käsitellään Microsoftin XNA peliohjelmointi teknologiaa sekä peliohjelmoinnin teoriaa. Työn tavoitteena on tuottaa kattava teoreettinen opas XNA:n käyttöön ja käyttöönottamiseen.

Opinnäytetyössä esitellään peliohjelmoinnin eri osa-alueita teoreettisesti ja tutkitaan samalla XNA:n etuja aikaisempiin toteutusmenetelmiin sekä XNA:n tulevaisuutta pelikehityksen kannalta.

Opinnäytetyössä pyritään tuottamaan kattava ohjeistus peliohjelmointiin ja XNA:n käyttöön peliohjelmoinnissa. Materiaalia pystytään käyttämään esimerkiksi opetustarkoitukseen tai lähdemateriaalina muihin ympäristöihin. Ohjeistuksen ohella kehitystehtävänä tuotetaan pienimuotoinen teknologiademo XNA:ta hyväksikäyttäen.

Kieli	Suomi
Asiasanat	XNA, pelikehitys, ohjelmointi
Säilytyspaikka	☒ Kajaanin ammattikorkeakoulun Kaktus-tietokanta ☒ Kajaanin ammattikorkeakoulun kirjasto

THESIS
ABSTRACT

School Business / Natural Sciences	Degree Programme Data Processing
Author(s) Teemu Heikkinen, Pavel Kauhanen, Antti Pikkarainen	
Title XNA as Game Development Environment	
Optional Professional Studies Programming	Instructor(s) Veli-Pekka Piirainen
	Commissioned by Veli-Pekka Piirainen / Kajaani University of Applied Sciences
Date 18.11.2008	Total Number of Pages and Appendices 102 + 1

The thesis discusses Microsoft XNA game programming technology, as well as the theory of general game programming. The objective was to produce a comprehensive theoretical guide on XNA and its use.

The thesis presents various areas of game programming in theory and the advantages of XNA compared to the earlier implementations. The thesis also analyses the role of XNA in game development in the future.

The thesis aimed to produce comprehensive documentation for game programming and for the use of XNA. The material can be used, for example as educational source material or in other situations. In addition, a small-scale technology demo was produced by using XNA.

Language of Thesis	Finnish
Keywords	Xna, game programming, programming
Deposited at	☒ Kaktus Database at Kajaani University of Applied Sciences ☒ Library of Kajaani University of Applied Sciences

SYMBOLILUETTELO

AMD Yksi maailman johtavia prosessori valmistajia.

CLR Common Language Runtime

Microsoft .NET Frameworkin virtuaalikone joka vastaa muistin, prosessien ja säikeiden hallinasta.

CTP Community Technogy Preview

Microsoftin käyttämä ohjelmistojen julkaisumuoto

DirectX Microsoftin kehittämä ohjelmointirajapinta grafiikalle.

Framework Ohjelmistokehys (myös sovelluskehys) tarkoittaa ohjelmistotuotetta, joka muodostaa rungon sen päälle rakennettavalle tietokoneohjelmalle. Ohjelmistokehys on ohjelmoinnin apuväline, jonka tarkoituksena on nopeuttaa uusien ohjelmistotuotteiden valmistusta.

GC Garbage Collector

Automaattinen ”roskien” kerääjä. Huolehtii ohjelmiston muistinkäytöstä.

High Level Shader language (HLSL)

Yleisesti käytössä oleva varjostimien ohjelmointikieli.

HUD Heads Up Display

Tapa esittää pelaajaan liittyviä tietoja pelin käyttöliittymässä häiritsemättä kuitenkaan pelaajan peliä. Käytetään usein kertomaan esimerkiksi pelaajan elinvoimaa, tavaroita ja pistemäärää.

IL Intermediate Language

Tavukoodi, joka käännetään ajon aikana prosessorille suoritettavaksi natiivi koodiksi.

I.S.R.O.T. Identity, Scale, Revolve, Orbit, Translate. 3D mallille suositeltu transformaatio sarja.

JIT Just-In-Time Compilation

Tavukoodin dynaaminen kääntäminen ajonaikana prosessorin ymmärtämään muotoon.

Matriisi Matemaattinen kaksiulotteinen taulukko joka sisältää alkioita.

Mesh Joukko verteksejä 3D-maailmassa.

MONO Mono on Novel-yrityksen tukema ohjelmistokehitysympäristö, jonka tavoitteena on luoda käyttöjärjestelmäriippumaton pohja Microsoftin kehittämälle .NET-arkkitehtuurille.

MVP Most Valuable Professional. Microsoftin myöntämä sertifikaatti ohjelmistokehittäjille.

Nvidia Yksi maailman johtavia näytönohjain valmistajia.

Pikseli Yksittäinen piste kuvaruudulla.

SDK Software development kit

Sprite Spriteiksi kutsutaan 3D-grafiikan sekaan lomitettuja litteitä 2D-hahmoja ja muita 2D-kuvia.

Varjostin Varjostin on joukko ohjelmiston käskyjä, mitä käytetään tuottamaan erilaisia piirtoefektejä. (eng. Shader)

Varjostin pyyhkäisy

Käsittää tietyn varjostimen yhden ajokerran.

Varjostin tekniikka

Tekniikka, jolla määrätään mitä varjostimia tietty varjostinajo käyttää.

Verteksi Yksittäinen kulmapiste 3D-maailmassa. (eng. vertex)

WAV	Microsoftin ja IBM:n kehittämä tiedostomuoto äänen tallentamiseen.
XACT	Cross-Platform Audio Creation Tool Äänten ohjelmointikirjasto ja työkalu joka julkaistaan Microsoftin DirectX SDK:n mukana
XBOX 360	Microsoftin pelikonsoli.

SISÄLLYS

1 JOHDANTO

XNA Framework on Microsoftin kehittämä peliohjelmointiin tarkoitettu kirjasto. XNA on rakennettu DirectX teknologian päälle ja se on pyritty tekemään mahdollisimman yksinkertaiseksi ohjelmoida, esimerkiksi erilaisten apukirjastojen kautta. XNA pelikehitysympäristön tarkoituksena on helpottaa pelikehitystä ja näin luoda uutta amatööripohjaa peliteollisuudelle. XNA mahdollistaa pelien toteutuksen samanaikaisesti PC:lle sekä Microsoftin XBOX 360 pelikonsolille. XNA pelejä voidaan tehdä käyttämällä C# ohjelmointikieltä, joka on aloittelevalle ohjelmoijalle helpommin lähestyttävissä kuin peliteollisuuden nykyisin paljon käyttämä C++. Microsoft on myös antanut XNA:n vapaampaan käyttöön, joka on mahdollistanut sen, että MONO-hankkeen yhteydessä on voitu tehdä oma XNA Frameworkin kaltainen kirjasto, jolloin XNA-pelit toimivat myös muissa käyttöjärjestelmissä kuin Microsoft Windowsissa.

Opinnäytetyön keskeisenä sisältönä on tarkastella peliohjelmointia, pelikehitystä ja näiden eri osa-alueita Microsoftin XNA kehitysympäristössä. Tavoitteena on selvittää, miten toteutetaan teknologiademo käyttäen Microsoftin XNA teknologiaa.

Opinnäytetyössä esitellään peliohjelmoinnin eri osa-alueita teoreettisesti ja tutkitaan samalla XNA:n etuja aikaisempiin toteutusmenetelmiin sekä XNA:n tulevaisuutta pelikehityksen kannalta. Opinnäytetyössä pyritään tuottamaan kattava ohjeistus peliohjelmointiin ja XNA:n käyttöön peliohjelmoinnissa. Materiaalia pystytään käyttämään esimerkiksi opetustarkoitukseen tai lähdemateriaalina muihin ympäristöihin. Ohjeistuksen ohella kehitystehtävänä tuotetaan pienimuotoinen teknologiademo XNA:ta käyttäen.

Opinnäytetyön tulisi vastata seuraaviin kysymyksiin. Miten XNA:lla voidaan toteuttaa pelejä? Miten XNA:n käyttäminen eroaa aikaisempiin toteutusmenetelmiin?

2 JOHDATUS XNA PELIKEHITYKSEEN

Microsoft aloitti XNA:n kehityksen 2000-luvun alkupuolella. Aluksi kehitys pidettiin hyvin salaisena. Vuonna 2004 Microsoft julkaisi XNA:n GDC (Game Developers Conference) -tapahtumassa ensimmäistä kertaa. XNA ei ole vain tavallinen luokkakirjasto kuten esimerkiksi DirectX rajapinta, vaan se sisältää lisäksi paljon erilaisia työkaluja ja Microsoft Visual Studion päälle rakennetun ohjelmointiympäristön. Mitään työkaluja tai osia ei julkaistu ennen vuotta 2006, vaan DirectX ohjelmoijat huomasivat ainoastaan XNA:n logon DirectX:n dokumentaation yläkulmassa vuosina 2004 - 2006. (Nitschke 2007, 3)

Tämä tarkoittaa sitä, että Microsoft työskenteli XNA:n parissa hyvin pitkään, mutta kehittäjät eivät tienneet mitä odottaa. Vuonna 2006 GDC:ssä Microsoft julkaisi XNA:n ensimmäisen build version nimikkeellä ”XNA Build March 2006 CTP”, jossa CTP:llä tarkoitetaan ”Community Technology Preview” käsitettä. (Nitschke 2007, 4 ; Wikipedia a)

Tämän jälkeen XNA:n kehitys eli hiljaiseloa ja vain Microsoftin henkilökunta ja DirectX MVP (Most Valuable Professional) kehittäjät saivat tietää tulevasta XNA rajapinnasta ja XNA Game Studio julkaisusta. Peliohjelmoijat saivat tietää XNA Game Studiosta, kun Microsoft julkisti ensimmäisen beta 1 version elokuun 30. päivä 2006. Ensimmäinen beta-versio sisälsi yhden valmiin aloituspelipaketin nimeltään ”Space Wars”, joka ei sisältänyt kovinkaan paljoa 3D -toimintoja. Monet kehittäjät ja harrastelijat kokeilivat XNA:ta ja kirjoittivat monia pieniä 2D -pelejä XNA:n Sprite-luokkien avulla. Vaikka XNA mahdollisti nopean 2D -pelien teon, kehittäjien oli hyvin vaikeaa kirjoittaa 3D grafiikan käsittelyyn liittyvää toiminnallisuutta. (Nitschke 2007, 4)

XNA Game Studio Express oli alkuperäisesti kohdistettu aloittelijoille, harrastelijoille ja opiskelijoille antaen heille mahdollisuuden kehittää nopeasti omia pelejä Windows käyttöjärjestelmälle ja XBOX 360 laitteistolle. Tämä ei kuitenkaan tarkoittanut etteivätkö ammattimaiset pelistudiot olisi voineet käyttää XNA:ta. (Nitschke 2007, 4)

Microsoft julkaisi vielä yhden beta version marraskuussa 2006 ennen lopullisen version julkaisua, joka tapahtui joulukuussa 2006. Lopullinen versio sisälsi Content Pipeline ominaisuuden ja monia muita uudistuksia. (Nitschke 2007, 4)

XNA on täysin ilmainen ja se antaa kehittäjille mahdollisuuden luoda pelejä sekä Windows alustalle että myös XBOX 360 laitteistolle yhtäaikaisesti. Mutta jos kehittäjä haluaa testata pelejään XBOX 360 konsolilla, hänen täytyy liittyä Creators Clubiin, jonka vuosittainen maksu on 99 dollaria. (Nitschke 2007, 4)

Opinnäytetyön kirjoitushetkellä uusin versio on 2.0 mikä sisältää mm. Visual Studio 2005 version käyttämisen XNA:n kanssa ja verkkopelien tekomahdollisuuden Windows ja XBOX 360 alustoille. (Wikipedia b)

Tulevassa 3.0 versiossa mahdollistetaan pelien teko myös Microsoftin valmistamiin Zune mediasoittimiin. (XNA Team Blog 2008)

2.1 Managed -koodi

Useimmilla kehitysalustoilla lähdekoodi käännetään suoraan natiiviksi binäärikoodiksi, joka ladataan ja ajetaan sillä prosessorilla mille se on tarkoitettu. Ajettava tiedosto näin ollen sisältää niin sanotut valmiit suoritusohjeet suorittimelle. (Hall 2008, 8)

Managed -ympäristössä koodi käännetään erilliselle välikielelle (Intermediate language eli IL). Välikoodi edustaa suoritusohjeita virtuaaliselle, alustariippumattomalle prosessorille. Tämä prosessori ei ole oikeasti olemassa, vaan se on tietokoneen muistissa sovelluksena. Jotta managed -koodi voitaisiin ajaa oikealla fyysisellä prosessorilla, täytyy koodin ja prosessorin välillä olla vielä erillinen prosessointi. Prosessoinnin hoitaa erillinen ajonaikainen ohjelma, joka toimii virtuaalisena prosessorina ja kääntää koodin viimeiseen ajettavaan muotoon. Tätä prosessia kuvataan kuviossa 2.1. (Hall 2008, 9)

Kuvio 2.1. Natiivi- ja Managed-koodin erot. (Hall 2008, 9)

Virtuaalisella prosessorilla emulointi kuulostaa tehottomalta verrattuna perinteiseen natiiviin koodiin. Mutta useimmat managed -koodin ajoympäristöt (Microsoftin mukaan lukien) tukevat ominaisuutta joka tunnettaan ”Just-in-Time”(JIT) kääntämisenä. JIT kääntäjä tuottaa managed -kääntäjän generoimasta välikoodista natiivia koodia, joka optimoidaan erikseen kohdeprosessorille juuri ennen kuin koodi ajetaan fyysisesti prosessorilla. Tämä tuo sinällään kiinnostavan näkökulman managed -koodiin. Mitä managed -koodilla pelin kehittäminen tarjoaa oikeasti pelikehittäjille ja mitä heillä ei olisi natiivilla C++ kielellä peliä tehdessä? (Hall 2008, 9)

Suurin hyöty tulee muistin käsittelyn helppoudesta. Tavallisissa ohjelmointikielissä, jotka kääntävät koodinsa natiiveiksi ajettaviksi tiedostoiksi, muistivuodot ovat tavallinen ongelma. On helppoa unohtaa vapauttaa varattu muisti etenkin, jos koodi on rakenteeltaan monimutkaista. (Hall 2008, 9)

Managed -koodissa kuten natiivissa koodissakin, muistia varataan käytön mukaan silloin kun sitä tarvitaan. Toisin kuin natiivissa koodissa, managed -koodin ajonaikainen ympäristö ylläpitää muistinvarauksia ja kaikkia viittauksia niihin. Silloin kun resurssiin ei enää viitata minkään olemassa olevan objektin kautta, resurssi vapautetaan automaattisesti. On silti joitain erikoistapauksia milloin muistivuotoja voi tapahtua. Käytännössä muistinhallinta tulee ilmaiseksi managed -ympäristössä. Tätä ominaisuutta kutsutaan roskien kerääjäksi (eng. Garbage Collector.) (Hall 2008, 10)

Toinen etu managed -koodissa on alustariippumattomuus. Koska koodi kohdistetaan virtuaaliselle prosessorille, sillä ei ole suoraa sidosta fyysiseen prosessoriin jolla se tullaan lopullisesti ajamaan. Tämä tarkoittaa, että managed -koodia pitäisi pystyä suorittamaan millä tahansa alustalla, joka tarjoaa ajon aikaisen kääntäjän sille. Teoriassa saman koodin tulisi toimia niin PC:llä, PDA -laitteilla ja pelikonsoleilla. Tämä on XNA:n avain ominaisuuteen, jotta sama koodi voidaan suorittaa sekä Windows että XBOX 360 alustoilla. Koska molemmat alustoista tukevat Common Language Runtime (CLR) ympäristöä ja XNA -luokkakirjastoa, pelit voivat siirtyä saumattomasti alustalta toiselle. (Hall 2008, 10)

Koodi, jota voidaan suorittaa monilla eri alustoilla, on itsessään jo hieno ominaisuus. Mutta ilman kunnollista pääsyä muihin komponentteihin kuin prosessori, ei koodilla voida tehdä mitään käytännöllistä. Pelkän ajoympäristön lisäksi managed -ympäristöt tarjoavat myös joukon erilaisia API -kirjastoja, jotka mahdollistavat pääsyn käyttöjärjestelmän ominaisuuksiin.

Nämä API -kirjastot mahdollistavat esimerkiksi tiedostojen luvun ja kirjoituksen, erilaisten käyttöliittymäkomponenttien käytön kuten napit, menut, sekä oheislaitteiden käytön kuten hiiri ja näyttö. (Hall 2008, 10)

2.2 XNA:n komponentit

Tietokoneen komponenttien käsittely erillisen abstraktin kerroksen läpi on tavallisesti paljon hitaampaa verrattuna niiden käsittelyyn suoraan natiivin koodin avulla. Tulostimen tai tiedostojärjestelmän kohdalla suorituskyky ei ole niin huomattavaa, koska laite tai kohde on itsessään hitaampi kuin sitä kutsuva koodi. Toiminnot odottavat suurimman osan ajastaan, että laite suorittaa sille annetun komennon. (Hall 2008, 10)

Mutta kun käsiteltävänä komponenttina on grafiikan piirtojärjestelmä, ylimääräisen kerroksen vaikutus on paljon suurempi. Miljoonien pikseleiden päivitys useasti sekunnissa vaatii paljon prosessoritehoa, vaikka sitä käytettäisiin natiivin koodin kautta. Onneksi on olemassa näytönohjain, jolle grafiikan prosessointia voidaan siirtää. Näytönohjaimen prosessori tunnetaan nimellä Graphics Processing Unit (GPU). (Hall 2008, 10-11.)

XNA -luokkakirjasto sisältää joukon kirjastoja, jotka tarjoavat pääsyn edistyneimpiin laitteiston ominaisuuksiin käyttäen teknologiaa nimeltä DirectX. DirectX on joukko ajureita sekä natiiveja API -kirjastoja, joilla näihin laitteisiin päästään käsiksi. DirectX, ja siten myös XNA, tarjoavat pääsyn muihinkin laitteisiin kuin näytönohjaimeen. Tuki syöttölaitteille ja esimerkiksi äänille on XNA:ssa toteutettu käyttämällä DirectX:n XINPUT ja XACT komponentteja. (Hall 2008, 11)

Sen lisäksi että XNA tarjoaa managed -koodilla pääsyn laitteistorajapintoihin ja DirectX kirjastoihin, se sisältää myös komponentin joka tunnetaan nimellä Content Pipeline. Content Pipelinen avulla hallitaan nimensä mukaisesti peliin tuotavaa ulkopuolista sisältöä. Tällaista sisältöä ovat tekstuurit, 3D objektit, musiikki, äänet ja muu pelin sisäinen tai ulkopuolinen data. (Hall 2008, 11)

Microsoftin .NET Framework:sta on tullut nopeasti johtava alusta tavallisten Windows -sovellusten toteutukseen. Vaikka XNA on erittäin riippuvainen .NET Frameworkin komponenteista, on XNA erityisesti suunniteltu ja optimoitu pelien kehitykseen. Koska XNA

kirjastot ovat erittäin helposti siirrettävissä Windows ja XBOX 360 alustojen välillä, voidaan pelit siis toteuttaa toiselle alustalle ja siirtää helposti toiselle alustalle ilman muutoksia tai hyvin pienillä muutoksilla. (Hall 2008, 11)

Kun tarvitaan tiedostojen, säikeiden tai taulukoiden käsittelyä ja hallintaa, käytetään apuna .NET Frameworkin kirjastoja. Kun soitetaan musiikkia, esitetään 3D tai 2D grafiikkaa ruudulla, käytetään XNA:ta. Nämä kaksi luokkakirjastoa siis täydentävät toisiaan erinomaisesti. (Hall 2008, 12)

2.3 C# Pelikehityksessä

CLR tukee monia ohjelmointikieliä, mutta ainoa virallinen tuettu kieli pelien kehitykseen XNA:ssa on C#. C# on lähellä Java ja C / C++ -ohjelmointikieliä. Mikäli joku näistä kielistä on tuttu, siirtyminen C# kieleen ei pitäisi olla vaikeaa. (Hall 2008, 13)

Peliteknologian kehityksessä C#:a käytetään useasti esimerkiksi pelikehitystä tukevien työkalujen kehitykseen. C# ohjelmointikieltä on myös käytetty useiden pelien tekoon, mutta suorituskykyä vaativissa sovelluksissa peliohjelmoijat ovat tyypillisesti käyttäneet natiiveja ohjelmointikieliä kuten C tai C++. Ennen XNA:n esittelyä C# ei ollut paras mahdollinen vaihtoehto pelien kehitykseen, sillä pelien täytyy toimia mahdollisimman nopeasti ja sulavasti. (Hall 2008, 13)

Vaikka C# on kielenä uusi tulokas peliohjelmoinnissa, on sitä kuitenkin käytetty peliobjektien ja tekoälyn skriptaukseen jo jonkin aikaa. Useasti pelilogiikan käsittelyssä käytetään skriptejä. Natiivia koodia käytetään vain niillä alueilla joissa suorituskyky on kriittisintä. (Hall 2008, 13)

Natiivi pelikoodi on taitavan ohjelmoijan erikoisalaa. Käyttämällä helposti omaksuttavaa skriptikieltä voidaan esimerkiksi kenttäsuunnittelijoille antaa toteutettavaksi yksinkertaista pelilogiikkaa ja kokeneemmat ohjelmoijat voidaan resursoida vaativien ja teknisten alueiden koodaamiseen. Hyötynä siitä, että suunnittelijat toteuttavat pelilogiikkaa on se, että tämä helpottaa ja parantaa myös huomattavasti suunnittelijoiden ja kehittäjien välistä kommunikaatiota. (Hall 2008, 13)

Internetin eri keskustelufoorumeilla on jatkuvasti keskusteluita C# ja C++ eroista. Useasti keskustelut ajautuvat helposti järjettömään väittelyyn kielien välillä. Väittelyllä ei sinällään ole mitään tekemistä C# -kielen kanssa, mutta se voi aiheuttaa huonoa mainetta, koska Java epäonnistui peliohjelmointiympäristönä muilla kuin mobiilialustoilla. Java ja C# ovat syntaksiltaan hyvin samanlaisia ja niissä molemmissa muistinkäyttö ja hallinta ovat automatisoituja. Samankaltaisia väittelyitä on tapahtunut aiemminkin, esimerkiksi kun C -kieli korvasi Assembly -konekielen ja C++ korvasi tavallisen C:n. Vielä tänäkin päivänä kun C++ on ollut saatavilla yli 20 vuotta, jotkut peliohjelmoijat silti käyttävät C:tä ja eivät ota kaikkea hyötyä C++:n kirjastoista. Esimerkiksi Quake ja Half-Life pelien lähdekoodit näyttävät enemmän C:ltä kuin C++:lta. (Nitschke 2007, 22)

Tulevaisuus tulee olemaan uusien kielien, jotka helpottavat koodin kirjoittamista. Useimmat isot pelitalot eivät kuitenkaan voi ottaa uutta teknologiaa heti käyttöön, koska ne voivat olla esimerkiksi keskellä suurta peliprojektia, jota ei voi helposti kääntää uudelle kielelle. (Nitschke 2007, 23)

3 XNA:N KÄYTTÖÖNOTTO JA OHJELMAMALLI

XNA luokkakirjasto on jaettu kolmeen keskeiseen osaan:

- XNA Graphic Engine, tiedostossa Microsoft.Xna.Framework.dll
- XNA Game Application Model, tiedostossa Microsoft.Xna.Frameword.Game.dll
- XNA Content Pipeline, tiedostossa Microsoft.Xna.Framework.Content.Pipeline.dll

Kaikki nämä luokkakirjastot ovat kirjoitettu C# ohjelmointikielellä. Näin ollen ne voidaan avata käyttäen työkaluja kuten Reflector, millä voidaan suoraan nähdä näiden luokkien sisäinen toiminta. Useimmat sisäiset funktiot ovat vain kutsuja DirectX luokkakirjastoihin. Reflector ohjelman voit hakea osoitteesta http://www.aisto.com/roeder/dotnet/. (Nitschke 2007, 6)

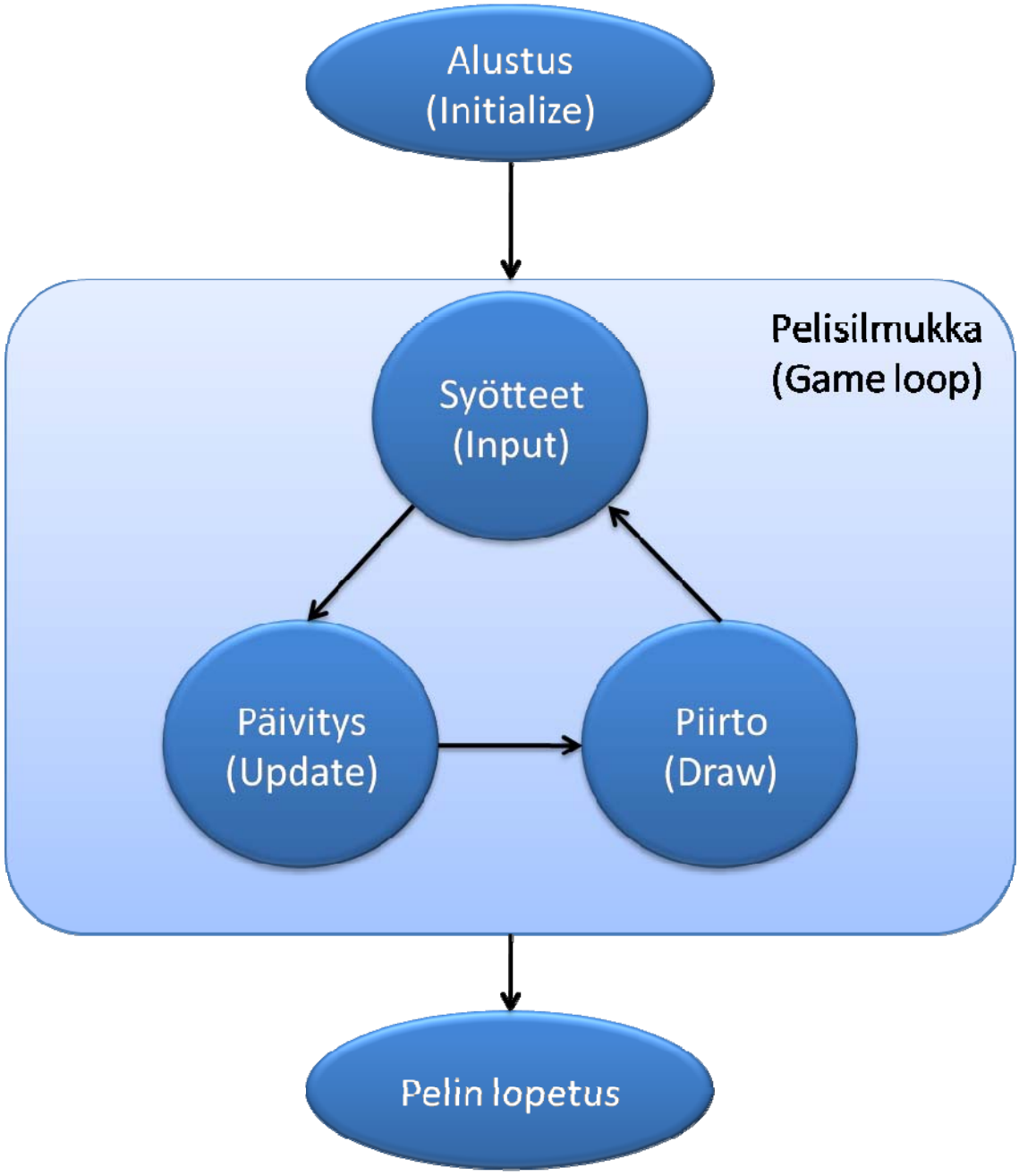
Jokainen XNA projekti käyttää Game -luokkaa, joka sisältää kaikki tärkeät pelikomponentit, grafiikanpiirtoon tarvittavat laitteistotiedot, ikkunan asetukset ja sisältötiedostojen hallinnan. Tarvittaessa tähän luokkaan voidaan lisätä myös syötteiden ja äänien hallinta. Pohjimmiltaan kaikki mitä peli tekee arkkitehtuurin ylimmillä tasoilla, johtaa johonkin Game -luokkaan tai ainakin johonkin komponenttiin mikä on yhteydessä Game -luokkaan. (Nitschke 2007, 6)

Seuraavat kolme metodia ovat tärkeimmät Game -luokassa:

- Initialize()
- Update(GameTime time)
- Draw(GameTime time)

Initialize -metodissa ladataan pelin sisältö, asetetaan kaikki tarvittavat aloitustiedot ja alustetaan ne. Jos halutaan noudattaa Microsoftin suunnittelumallia, kaikki sisällön lataaminen tehdään **LoadGraphicsContent** -metodissa. **Update** -metodia kutsutaan jokaisen kuvaruudun (frame) piirron yhteydessä. Tällöin päivitetään pelihahmojen sijainteja, päivitetään peliaikaa, käsitellään erilaisia syötteitä sekä ääniä ja lopuksi kutsutaan **Draw** -metodia jokaisen framen yhteydessä piirtämään kaikki tarvittava grafiikka ruudulle (Kuvio 3.1). On tärkeää, että pelin logiikka suoritetaan omassa lohkossaan riippumattomana piirtologiikasta. Esimerkiksi Win-

dows alustalla käyttäjä voisi painaa Alt + Tab näppäinyhdistelmää tai pienentää peli-ikkunan, jolloin **Draw** -metodia ei tarvitse enää kutsua, mutta pelin toiminta siltikin jatkuu taustalla. Tämä on erittäin tärkeää esimerkiksi verkkopeleissä, joissa täytyy pitää huolta, että pelaaja on synkronoitu serverin ja muiden pelaajien kanssa. (Nitschke 2007, 7)



Kuvio 3.1. Pelisilmukka.

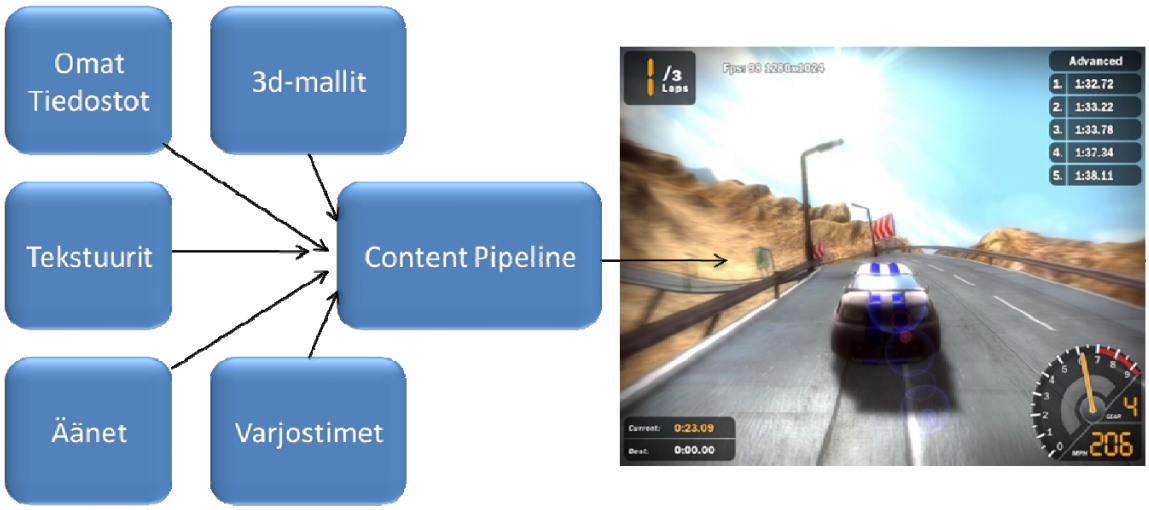
Lisäksi kehittäjä voi lisätä **GameComponent** -luokkia peliinsä, joilla on myös Update- ja Draw -metodit. Kumpiakin näistä metodeista kutsutaan automaattisesti pelin Update- ja Draw -metodeista. Alkuperäisesti Microsoft tahtoi, että kehittäjät rakentaisivat ja lisäisivät GameComponent luokkia erillisen suunnittelutilan kautta, joka oli mukana ensimmäisessä XNA Game Studion beta versiossa (30 Elokuuta, 2006). Suunnittelutila poistettiin myöhemmin, koska se ei toiminut hyvin ja XBOX 360 -alusta ei sitä tukenut. (Nitschke 2007, 7)

GameComponent luokkien tarkoituksena on, että koodia voitaisiin helposti käyttää uudelleen tehokkaasti ja niiden liittäminen peliin olisi helppoa. Esimerkkinä mm. ruudunpäivitystä laskeva mittari, 3D ympäristön tai taivaan piirto pelaajalle. (Nitschke 2007, 8)

Yksi Microsoftin odotuksista ohjelmamallin osalta on se, että kehittäjät ja ohjelmoijat voivat jakaa pelikomponenttejaan helposti muun peliyhteisön kanssa. Tämä kohentaa huomattavasti XNA:n yhteisöllistä näkökantaa. (Nitschke 2007, 8)

3.1 Content Pipeline

Content Pipeline on luokkakirjasto, jota käytetään tuomaan, kääntämään ja lataamaan pelin sisältötiedostot kuten tekstuurit, 3D-mallit, varjostimet ja äänitiedostot. (Kuvio 3.2). Se pienentää huomattavasti oman koodin kirjoituksen tarvetta siihen, että omat grafiikat, 3D mallitiedostot ja varjostimet saataisiin tuotua peliin. Esimerkiksi jos peliprojektin tuodaan 3D mallitiedosto, joka käyttää kahta tekstuuria ja erityistä varjostinta, Content Pipeline osaa käsitellä mallitiedoston ja automaattisesti etsiä ja lisätä tarvittavat tekstuurit ja varjostimet. Sisältö tuodaan ja käännetään binääriseen muotoon ja tehdään kaikki tarvittava automaattisesti. (Nitschke 2007, 8)



Kuvio 3.2. Content Pipeline.

Content Pipeline ei ole vain yksi itsenäinen luokka, vaan se sisältää viisi erilaista luokkakirjastoa:

- **Microsoft.Xna.Framework.Content.Pipeline.dll**, joka sisältää perusfunktiot Content Pipelinea varten.

- **Microsoft.Xna.Framework.Content.Pipeline.EffectImported.dll**, jota käytetään kääntämään ja tuomaan varjostimia.

- **Microsoft.Xna.Framework.Content.Pipeline.FBXImporter.dll**, joka on isoin luokkakirjastoista sisältäen paljon koodia .fbx 3D-mallien tuontia varten. Se tukee monia ominaisuuksia kuten 3D-mallien luita ja päällysteitä.

- **Microsoft.Xna.Framework.Content.Pipeline.TextureImporter.dll**, jota käytetään tuomaan tekstuuritiedostoja peliin. Nämä tiedostot voivat olla esim. .dds-tiedostoja, jotka ovat jo valmiiksi DirectX formaatissa (paras formaatti tekstuureille ja se tukee laitteistopohjaista pakkausta). Myös .png, .jpg, .bmp, ja .tga tiedostot ovat tuettuja. kaksiulotteiset sprite tiedostot ovat myös pelkästään tekstuureita ja niitä käytetään yleensä pakkaamattomassa 32 bittisessä formaatissa.

- **Microsoft.Xna.Framework.Content.Pipeline.XImporter.dll**, joka mahdollistaa .x 3D-mallitiedostojen tuonnin peliin. Samaa tiedostomuotoa on käytetty myös moniin DirectX ohjelmiin ja esimerkkeihin. (Nitschke 2007, 9)

Peliprojekti itsessään ei tarvitse näitä luokkakirjastoja, niitä käytetään vain rakentamaan ja kääntämään pelin sisältö .xnb (XNA Binary) tiedostoksi. Tämä tekee tiedostojen levityksestä helpompaa, koska kehittäjän ei tarvitse huolehtia pelin sisältötiedostoista. Näin ollen on helpompi varmistaa, että kaikki sisältötiedostot ovat saatavilla, kun peli käynnistetään. Ei ole suositeltavaa muokata .xnb tiedostoja käsin, koska ne ovat suoraan käännetyssä muodossa kuten mikä tahansa .exe tiedosto. Dataa ei voida myöskään kääntää takaisin tekstuureiksi, malleiksi tai varjostimiksi. Tiedostot eroavat myös alustasta riippuen. Näin ollen .xnb tiedostot ovat erilaisia Windowsilla ja XBOX 306:lla, vaikka pelin lähdekoodi ja sisältötiedostot olisivatkin täysin samanlaisia. (Nitschke 2007, 9)

Vaihtoehtoisesti kehittäjä voi tehdä tarvittaessa oman muunnellun sisällön tuojan, mikä mahdollistaa muiden tiedostoformaattien kääntämisen .xnb tiedostoksi. (Nitschke 2007, 9)

4 3D -MATEMATIIKKA

Vektorit ovat pelimaailman peruspilareita ja niitä voidaan hyödyntää monissa eri asioissa kuten esimerkiksi animaatioiden luonnissa, törmäyksen tunnistuksessa, valaistuksessa sekä monissa muissa asioissa. Koska vektoreita käytetään paljon, on oleellista ymmärtää niihin liittyvä matematiikka. Matriisimatematiikka on osa lineaarialgebraa ja sen ymmärtämisestä on hyötyä 3D ohjelmoinnissa. (Cawood & McGee 2007, 194, 208 ; Hall 2008, 103, 112-113)

4.1 Vektorit

Vektori ilmaisee yksittäisen pisteen esimerkiksi pelimaailman koordinaatistossa. Vektori itsessään sisältää niin monta koordinaattia kuin pelimaailmassa on ulottuvuuksia. Kaksiulotteisessa pelimaailmassa vektori sisältää arvot X ja Y koordinaatit, kun taas kolmiulotteisessa pelimaailmassa näiden lisäksi on käytössä Z eli syvyys. Neliulotteinen vektori taas sisältää edellisten lisäksi myös W -koordinaatin. Neliulotteista vektoria voidaan käyttää esimerkiksi värimäärityksessä, jolloin W -koordinaattia määrää läpinäkyvyyden ja X, Y sekä Z sisältävät arvot punaiselle, vihreälle ja siniselle. (Cawood & McGee 2007, 194 ; Hall 2008, 103)

XNA -kirjasto tarjoaa kolme luokkaa vektoreille, jotka ovat Vector2, Vector3 ja Vector4. Nämä luokat ovat siis järjestyksessä kaksi-, kolmi- ja neliulotteisille vektoreille. Jokainen luokka sisältää samanlaiset metodit vektorien matemaattista käsittelyä varten. Koska luokat sisältävät eri määrin ulottuvuus pisteitä, niiden sisäinen käsittely poikkeaa toisistaan. Matemaattiset perusoperaatiot kuten yhteenlasku, vähennyslasku sekä skaalaus ovat kaikille vektori luokille samanlaisia. Näiden lisäksi luokat tarjoavat myös metodeja monimutkaisempia operaatio varten. Näitä ovat esimerkiksi vektorin pituuden laskeminen, pintaan kohtisuorassa olevan vektorin laskeminen ja kahden vektorin välisen kulman laskeminen. (Cawood & McGee 2007, 194 ; Hall 2008, 103)

Vektorien yhteenlasku on oleellinen osa monissa pelialgoritmeissa. Kuviossa 4.1 on esimerkki kahden kolmiulotteisen vektorin yhteenlaskusta. Tulos vektorin vC arvoiksi tulee X=9, Y=1 ja Z=0. Eli yksinkertaisesti kaikkien vastaavien koordinaattien arvot lasketaan yhteen. (Cawood & McGee 2007, 194-195.)

```
Vector3 vA = new Vector3(5.0f, 3.0f, 0.0f); // Vektori A
Vector3 vB = new Vector3(4.0f, -2.0f, 0.0f); // Vektori B
Vector3 vC; //Tulos vektori C

vC = vA + vB;
```

Kuvio 4.1. Vektorien yhteenlasku

Vektorien vähennyslasku on keskeistä aina silloin, kun halutaan laskea kahden vektorin välinen etäisyys. Kuvion 4.2 esimerkissä vektorista A(5, 3, 0) vähennetään vektori B(4, -2, 0). Vähennyslaskun jälkeen tulosvektorin vC arvoiksi tulee X=1, Y=5 ja Z=0. (Cawood & McGee 2007, 196)

```
Vector3 vA = new Vector3(5.0f, 3.0f, 0.0f); // Vektori A
Vector3 vB = new Vector3(4.0f, -2.0f, 0.0f); // Vektori B
Vector3 vC; //Tulos vektori C

vC = vA - vB;
```

Kuvio 4.2. Vektorien vähennyslasku

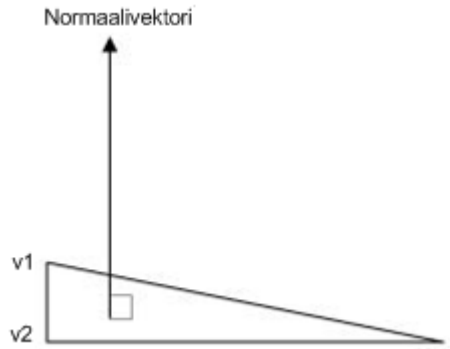
Vektorin skaalaus tarkoittaa vektorin koon muutosta. Tämä tapahtuu joko kertomalla tai jakamalla haluttu vektori. XNA:n vektori luokkia skaalatessa arvon pitää olla liukuluku. Skaalausta voidaan käyttää apuna esimerkiksi animoinnissa. Kuviossa 4.3 on esimerkki vektorin skaalauksesta kertoen sekä jakaen. Ensimmäisen skaalauksen vektorin v arvoksi saadaan X=18, Y=2 ja Z=0. Toisen skaalauksen jälkeen vektori on alkutilassaan, koska molemmilla skaalauksilla käytettiin samaa kerrointa. (Cawood & McGee 2007, 196-197.)

```
Vector3 v = new Vector3(9.0f, 1.0f, 0.0f);
float fScale = 2.0f; //Kerroin liukulukuna
v *= fScale; //Skaalaus kertomalle
v /= fScale; //Skaalaus jakamalla
```

Kuvio 4.3. Vektorin skaalaus

4.1.1 Normaalivektori

Normaalivektori on erikoisvektori, jota käytetään mm. valaistuksessa tai kameran käsittelyssä. Normaalivektori on vektori, joka on kohtisuorassa eli 90 asteen kulmassa tasaiseen pintaan nähden. Normaalivektori ilmaisee suuntaa, joten sen sijainti ei ole oleellinen. Kuviossa 4.4 on kahden vektorin v1 ja v2 muodostaman pinnan normaalivektori. (Cawood & McGee 2007, 197-198. ; Hall 2008, 105 - 106.)



Kuvio 4.4. Normaalivektori

Normaalivektoria, joka lasketaan kahden vektorin muodostamasta pinnasta, kutsutaan ristituloksi (cross product). Alla on matemaattinen kaava jolla ristitulo lasketaan.

$$A \times B = (A_y*B_z - A_z*B_y, A_z*B_x - A_x*B_z, A_x*B_y - A_y*B_x)$$

Vector3 -luokan Cross -metodi laskee ristitulon. Parametreinä metodille annetaan kaksi vektoria, jotka muodostavat pinnan. Kuviossa 4.5 on esimerkki sen käytöstä. Vektori v1 on vektorien vA ja vB muodostaman pinnan normaalivektori. Parametreinä annettujen vA ja vB -vektorien järjestys määrää niistä muodostuvan pinnan ”yläpuolen” ja vaikuttaa oleellisesti tulokseen. (Cawood & McGee 2007, 197-198.)

```
//Alustetaan vektorit
Vector3 vA = new Vector3(1, 2, 0);
Vector3 vB = new Vector3(0, 3, 0);

//Lasketaan cross product
Vector3 v1 = Vector3.Cross(vA, vB);
```

Kuvio 4.5. Cross -metodi

4.1.2 Vektorin normalisointi

Vektorin normalisointi tarkoittaa vektorin skaalausta yksikkövektoriksi. Yksikkövektori on vektori, joka on skaalattu siten että sen pituus on 1 ja sen koordinaattien arvot ovat väliltä -1 ja 1. Yksikkövektoria käytetään esimerkiksi ominaisuuksien vertailussa kuten, nopeus tai suunta. Tällöin pituudella ei ole merkitystä, mutta suhteellinen muutos X, Y, Z koordinaattien välillä on merkitsevä. Yksikkövektori lasketaan jakamalla jokainen vektorin koordinaatti vektorin pituudella. Vektorin pituus taas lasketaan käyttäen Pythagoraan lausetta, jossa koor-

dinaattien arvot korotetaan toiseen potenssiin, lasketaan yhteen ja summasta otetaan neliöjuuri. Alla on kaava vektorin pituuden laskemiseen. (Cawood & McGee 2007, 200-201.)

$$vektorinpituus = \sqrt{X^2 + Y^2 + Z^2}$$

Vektorien pituuksien laskemiseen voidaan käyttää Vector -luokkien Length -metodia. Yksikkövektori taas voidaan suoraan laskea käyttämällä Vector -luokkien Normalize -metodia. Kuviossa 4.6 on esimerkki metodien käytöstä. (Cawood & McGee 2007, 201-203.)

```
Vector3 v = new Vector3(7.0f, -2.0f, 0.0f);
float fLength = (float)v.Length(); //Vektorin pituuden laskeminen
Vector3 vUnit = Vector3.Normalize(v); //Yksikkövektorin laskeminen
```

Kuvio 4.6. Vektorin laskeminen

4.1.3 Pistetulo

Pistetuloa (dot product) käytetään laskettaessa kahden vektorin välinen kulma. Sitä voidaan käyttää mm. laskettaessa lentoratoja, heijastuskulmia tai valon voimakkuutta. Pistetulo voidaan laskea matemaattisella kaavalla, mutta XNA -kirjaston Vector -luokkien Dot -metodilla voidaan myös helposti laskea pistetulo. Kuviossa 4.7 on esimerkki Dot -metodin käytöstä. (Cawood & McGee 2007, 203-205.)

```
//Vektorien alustus
Vector3 vA = new Vector3(1.0f, 0.0f, 0.0f);
Vector3 vB = new Vector3(4.0f, 2.0f, 0.0f);

//Yksikkö vektorien laskeminen
Vector3 vUnitA = Vector3.Normalize(vA);
Vector3 vUnitB = Vector3.Normalize(vB);

//Dot productin laskeminen
float fDotProduct = Vector3.Dot(vA, vB);
```

Kuvio 4.7. Dot -metodi

4.2 Matriisit

Matriiseja voidaan käyttää esimerkiksi tiedon, verteksien tai objektin muunnoksen säilöntään. 3D -objekti on kokoelma pisteitä ja niitä voitaisiin käsitellä yksitellen, mutta matriisien avulla

niiden käsittelyä voidaan yksinkertaistaa. Matriisi on kokoelma rivejä ja sarakkeita, joiden sisältämien tietojen avulla 3D objektille voidaan tehdä muunnoksia (transformaatio), joita ovat mm. skaalaus, kierto tai siirto. Matriisi voi sisältää yhden tai useampia muunnoksia. XNA:n tarjoama matriisiluokka sisältää neljä riviä ja neljä saraketta. Jokainen yksittäinen solu sisältää liukuluvun ja solun arvo saadaan rivin ja sarakkeen perusteella. Esimerkiksi ensimmäinen solu on rivillä yksi ja sarakkeessa yksi. (Cawood & McGee 2007, 208 ; Hall 2008, 112-113.)

4.2.1 Matriisien kertolasku

Matriisien kertolasku suoritetaan kertomalla ensimmäisen matriisin rivit toisen matriisin sarakkeilla. Jotta kertolasku voidaan suorittaa, ensimmäisessä matriisissa täytyy olla sarakkeita yhtä monta kuin toisessa rivejä. XNA avulla matriiseja voidaan kuitenkin kertoa suoraan keskenään kuten kuviossa 4.8. (Cawood & McGee 2007, 208-211.)

```
// Luodaan tarvittavat matriisit
Matrix A = new Matrix();
Matrix B = new Matrix();
Matrix C = new Matrix();

// Alustetaan matriisi A. Muiden solujen arvoksi jää 0
A.M11 = 2.0f; A.M12 = 1.0f; A.M13 = 0.0f; A.M14 = 0.0f;

// Alustetaan matriisi B
B.M11 = 2.0f; B.M12 = 1.0f; B.M13 = 3.0f; B.M14 = 1.0f;
B.M21 = 1.0f; B.M22 = 2.0f; B.M23 = 4.0f; B.M24 = 1.0f;
B.M31 = 0.0f; B.M32 = 3.0f; B.M33 = 5.0f; B.M34 = 1.0f;
B.M41 = 2.0f; B.M42 = 1.0f; B.M43 = 2.0f; B.M44 = 1.0f;

// Kerrotaan matriisi A matriisilla B. Tulos saadaan matriisiin C
C = A * B;

// Matriisin C solujen arvoiksi tulee
// 5.00     4.00        10.00       3.00
// 0.00     0.00        0.00        0.00
// 0.00     0.00        0.00        0.00
// 0.00     0.00        0.00        0.00
```

Kuvio 4.8. Matriisien kertolasku

4.2.2 Transformaatiomatriisit

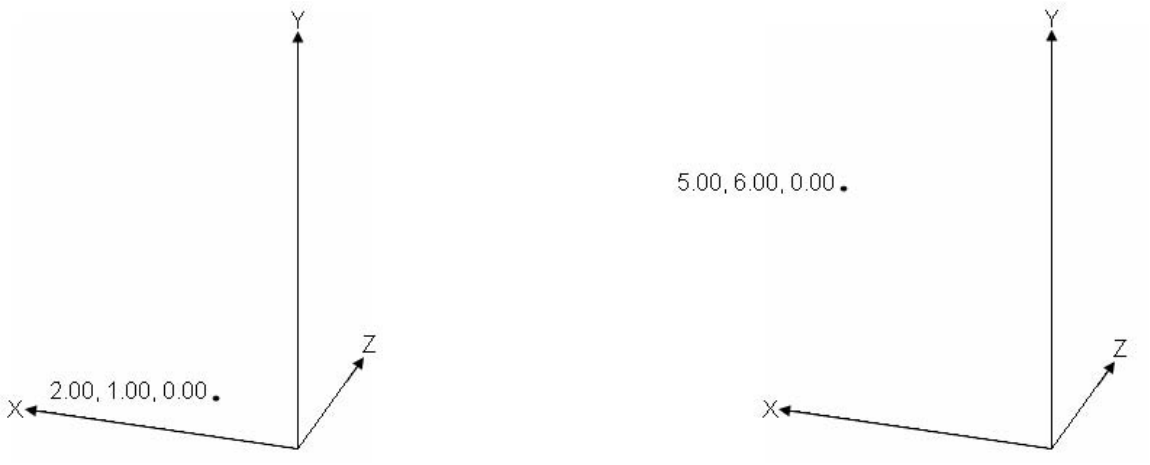
Matriiseja käytetään muuntamaan verteksejä, kun piirretään 3D -malleja tai yksinkertaisempia muotoja. Transformaatiomatriiseja käytetään verteksijoukon skaalauksessa, kierrossa ja siirrossa. Kun objektille tehdään useampi muunnos, niiden järjestyksellä on merkitystä. Jos muunnokset tehdään väärässä järjestyksessä, lopputulos voi olla väärä. Perusjärjestyksenä on yleensä: ensin skaalaus, seuraavana kierto ja viimeisenä siirto. Tämä johtuu siitä, että jos esimerkiksi kierrettäessä objekti ei ole origossa se kiertää origoa eikä kierry oman akselinsa ympäri. XNA käyttää niin sanottua oikeakätistä koordinaatistoa, jonka vuoksi muunnoksia laskettaessa muunnettavan matriisin pitää olla vasemmalla ja transformaatiomatriisin oikealla operaattoriin nähden. (Cawood & McGee 2007, 213 ; Hall 2008, 112-113.)

Siirtomatriisia käytetään silloin, kun objektin paikkaa täytyy muuttaa. Esimerkiksi pelimaailman alustuksessa voidaan 3D -mallit asettaa oikeille paikoilleen käyttäen siirtomatriisia. Siirtomatriisi sisältää arvot transformaatioita varten X, Y ja Z akseleille. Lävistäjän arvoina on kaikissa kohdissa 1. Kun siirretään esimerkiksi vektoria, jolla on arvot X, Y ja Z koordinaateille täytyy neljännen koordinaatin eli W:n arvoksi asettaa 1 jotta tuloksesta tulee tarkka. Alla esimerkki siirtomatriisista:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ X & Y & Z & 1 \end{pmatrix}$$

Ajatellaan, että verteksi pisteessä X=2, Y=1 ja Z=0 kerrotaan siirtomatriisilla, jonka arvoiksi on asetettu X=3, Y=5 ja Z=0. Näiden tietojen pohjalta voidaan päätellä, että laskutoimituksen jälkeen verteksi on siirtynyt X akselilla 3 mittayksikköä positiiviseen suuntaan ja Y akselilla 5 mittayksikköä positiiviseen suuntaan. Alla esimerkit laskutoimituksesta sekä kuva (Kuvio 4.9), jossa verteksi ennen ja jälkeen laskutoimituksen. Huomaa että laskutoimituksessa verteksin W arvo on 1. (Cawood & McGee 2007, 213-215.)

$$\begin{pmatrix} 2 & 1 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 5 & 0 & 1 \end{pmatrix}$$



Kuvio 4.9. Verteksi ennen ja jälkeen siirtomatriisitransformaation

Käyttäen XNA:n tarjoamaa CreateTranslation -metodia yllä oleva laskutoimitus voidaan suorittaa hyvin yksinkertaisesti (Kuvio 4.10). Metodi ottaa vastaan attribuutteina arvot X, Y ja Z akseleille ja näiden pohjalta muodostaa siirtomatriisin automaattisesti. (Cawood & McGee 2007, 215 - 216.)

```
//Luodaan matriisi verteksille ja alustetaan se
Matrix A = new Matrix();
A.M11 = 2.0f; A.M12 = 1.0f; A.M13 = 0.0f; A.M14 = 1.0f;

//Luodaan siirtomatriisi
Matrix B = Matrix.CreateTranslation(3.0f, 5.0f, 0.0f);

//Suoritetaan siirto
Matrix C = A * B;
```

Kuvio 4.10. Siirtomatriisi

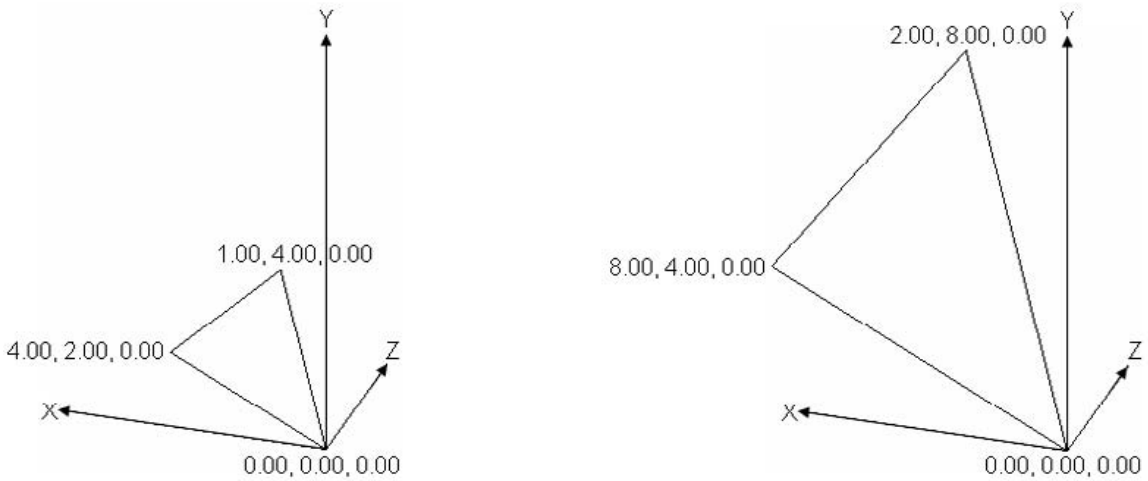
Skaalausmatriisia käytetään silloin, kun objektin kokoa täytyy muuttaa. Esimerkiksi 3D -malleja täytyy skaalata, jos kaikki käytetyt mallit eivät ole oikeassa suhteessa toisiinsa. Skaalausmatriisi sisältää tiedot objektin koon muutokseen X akselilla A yksikön verran, Y akselilla B yksikön verran ja Z akselilla C yksikön verran. A, B ja C arvot asetetaan matriisin lävistäjälle ja lävistäjän viimeiseen paikkaan tulee arvoksi 1 muut matriisin arvot ovat 0. Alla on esimerkki skaalausmatriisista. (Cawood & McGee 2007, 216)

$$\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & B & 0 & 0 \\ 0 & 0 & C & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ajatellaan, että on kolmio jonka kärkipisteet ovat (0,0,0), (1,4,0) ja (4,2,0). Jos tämän kolmion koko halutaan kaksinkertaistaa, voidaan käyttää skaalausmatriisia jonka kaikkien akselien skaalaus arvoksi on asetettu 2. Kun kolmiomatriisi ja skaalausmatriisi kerrotaan keskenään, tuloksena saadaan kolmio, joka on kaksi kertaa alkuperäistä suurempi. Alla on esimerkki kolmion skaalauksesta. Vasemmalla on matriisi, joka sisältää kolmion kärkipisteet ja oikealla on skaalausmatriisi. (Cawood & McGee 2007, 216)

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Kuvassa (Kuvio 4.11) on havainnollistettu kolmion koon muutos ennen ja jälkeen skaalauksen.



Kuvio 4.11. Kolmio ennen ja jälkeen skaalauksen

XNA:n CreateScale -metodi automatisoi tämän laskutoimituksen. Metodi ottaa vastaan attribuutteina arvot X, Y ja Z akselien skaalaukseen ja palauttaa niiden pohjalta luodun skaalausmatriisin. Kuviossa 4.12 on esimerkki metodin käytöstä. (Cawood & McGee 2007, 218)

```
//Luodaan matriisi kolmiolle ja alustetaan se
```

```
Matrix A = new Matrix();
A.M11 = 0.0f; A.M12 = 0.0f; A.M13 = 0.0f; A.M14 = 0.0f;
A.M21 = 1.0f; A.M22 = 4.0f; A.M23 = 0.0f; A.M24 = 0.0f;
A.M31 = 4.0f; A.M32 = 2.0f; A.M33 = 0.0f; A.M34 = 0.0f;
A.M41 = 0.0f; A.M42 = 0.0f; A.M43 = 0.0f; A.M44 = 0.0f;

//Luodaan skaalausmatriisi
Matrix B = Matrix.CreateScale(2.0f, 2.0f, 2.0f);

//Suoritetaan skaalaus
Matrix C = A * B;
```

Kuvio 4.12. Skaalausmatriisi

Kiertomatriiseja käytetään muuntamaan verteksijoukon kulmaa. Kiertomatriiseja voidaan hyödyntää esimerkiksi silloin, kun halutaan pyörittää 3D mallia kuten rengasta. Kiertomatriiseja on kolme, yksi jokaista akselia kohden.

X akselin kiertomatriisi:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Y akselin kiertomatriisi:

$$\begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Z akselin kiertomatriisi:

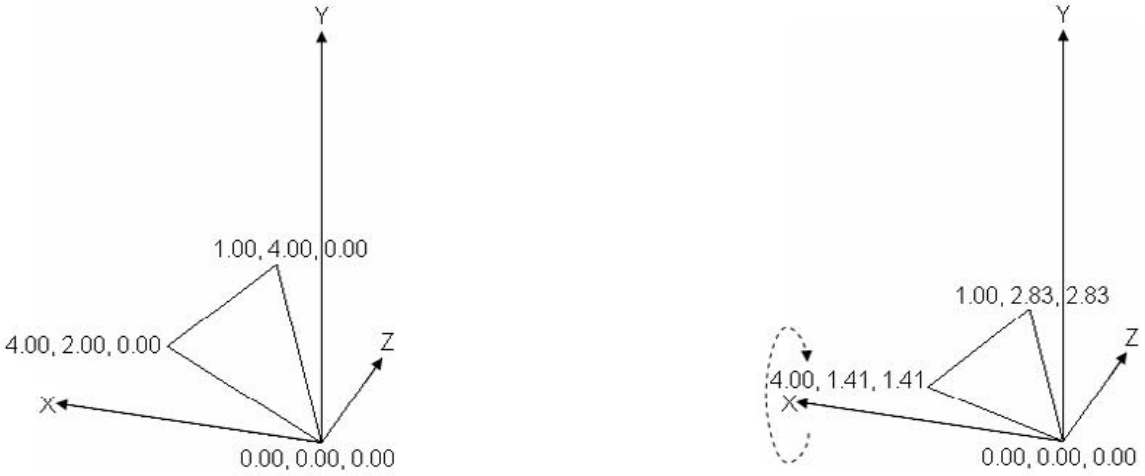
$$\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Kaikkien akselin kiertomatriisit ovat siis periaatteessa melkein samankaltaisia. Matriisien lävistäjän arvoina on 1, jollei siihen ole asetettu kulman arvoa radiaaneina. Kiertomatriisien käyttö on kaikkien akselien tapauksessa samanlainen: muunnettava matriisi kerrotaan kiertomatriisilla. Alla olevassa esimerkissä kolmiota, jonka kulmat ovat (0,0,0), (1,4,0) ja (4,2,0), muunnetaan kiertämällä sitä X akselin suhteen 45 astetta. Esimerkkiä on havainnollistettu kuviossa 4.13.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\pi/4) & \sin(\pi/4) & 0 \\ 0 & -\sin(\pi/4) & \cos(\pi/4) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Laskutoimituksen tulokseksi tulee seuraavanlainen matriisi:

$$\begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 \\ 1.00 & 2.83 & 2.83 & 0.00 \\ 4.00 & 1.41 & 1.41 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 \end{pmatrix}$$



Kuvio 4.13. Kolmion muunto X akselin kiertomatriisilla

XNA:n Matrix -luokasta löytyvien metodien avulla voidaan kiertomatriisit luoda helposti eri akseleille. Jokaiselle akselille löytyy oma metodinsa. Metodit ottavat parametrina vastaan liukuluvun, joka kertoo kierron kulman radiaaneina. Metodit ovat CreateRotationX, CreateRotationY ja CreateRotationZ. Kuviossa 4.14 on esimerkki CreateRotationX -metodin käytöstä. (Cawood & McGee 2007, 218-223.)

```
//Luodaan matriisi kolmiolle ja alustetaan se
Matrix A = new Matrix();
A.M11 = 0.0f; A.M12 = 0.0f; A.M13 = 0.0f; A.M14 = 0.0f;
A.M21 = 1.0f; A.M22 = 4.0f; A.M23 = 0.0f; A.M24 = 0.0f;
A.M31 = 4.0f; A.M32 = 2.0f; A.M33 = 0.0f; A.M34 = 0.0f;
A.M41 = 0.0f; A.M42 = 0.0f; A.M43 = 0.0f; A.M44 = 0.0f;

//Luodaan 45 asteen kiertomatriisi
Matrix B = Matrix.CreateRotationX((float)(Math.PI / 4.0));

//Suoritetaan kierto X akselilla
Matrix C = A * B;
```

Kuvio 4.14. Kiertomatriisi

4.2.3 Yksikkömatriisi

Yksikkömatriisilla kerrottaessa ei tapahdu muutosta, vaan tulos on sama kuin alkuperäinen matriisi. Vaikka tämä tuntuu turhalta, on yksikkömatriisi osa suositeltua I.S.R.O.T. (Identity, Scale, Revolve, Orbit, Translate) muunnosjärjestystä. Yksikkömatriisilla varmistetaan, että maailmamatriisi on oikein alustettu ennen kuin muita transformaatiomatriiseja käytetään. Yksikkömatriisia käytetäänkin oletuksena maailmamatriisin alustuksessa. Yksikkömatriisin lävistäjän arvoina on 1 ja muissa soluissa 0. XNA:n Matrix luokan Identity -metodilla voidaan luoda yksikkömatriisi kuten kuviossa 4.15 on esitetty. Alla yksikkömatriisi: (Cawood & McGee 2007, 218-223.)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

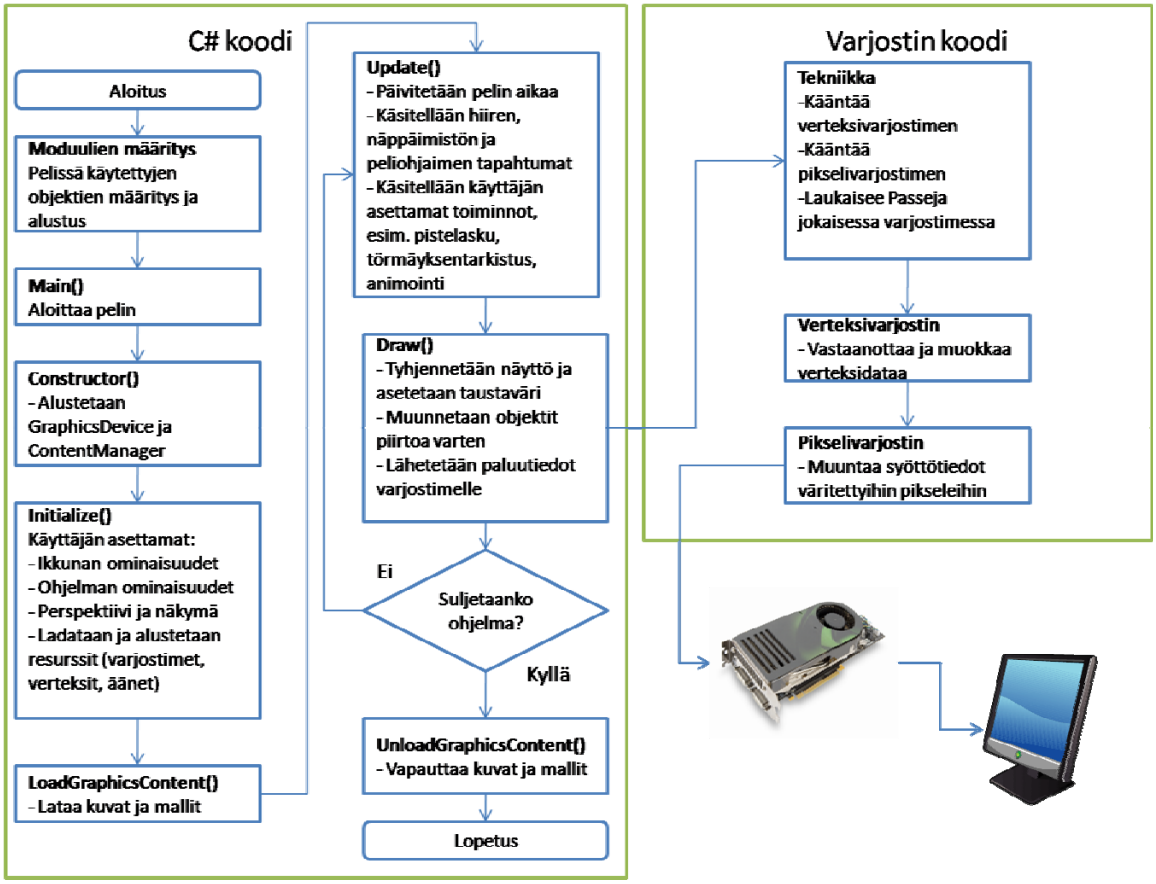
```
//Luodaan matriisi
Matrix matIdent;

//Alustetaan matriisi yksikkömatriisiksi
matIdent = Matrix.Identity;
```

Kuvio 4.15. Yksikkömatriisi

5 XNA GRAFIIKKAOHJELMOINTI

XNA tarjoaa paljon työkaluja, joilla grafiikkaohjelmointia on pyritty helpottamaan kehittäjän näkökulmasta katsottuna. Grafiikkaohjelmointi mahdollistaa erilaisten elementtien piirtämisen näytölle. XNA tarjoaa kehittäjälle paljon erilaisia peruselementtejä, joiden hallinta on helppoa ja yksinkertaista. Peliprojektin eli peli-ikkunan luonti on helppoa, koska XNA tekee kaikki tarvittavat alustukset valmiiksi, joten kehittäjä pääsee suoraan ohjelmoimaan peliä. XNA tarjoaa perustoiminnot, joilla luodaan, piirretään ja päivitetään peli-ikkunaa ja graafisia peruselementtejä. Peli-ikkunan luonti, päivitys ja piirtotoiminnot ovat esitettynä kuviossa 5.1 näkyvässä vuokaaviossa. XNA:ssa kaikki piirtäminen tapahtuu samaa peruslogiikkaa käyttäen, eli samaa piirtologiikkaa käytetään sekä 3D -mallien että 2D -kuvien piirrossa.



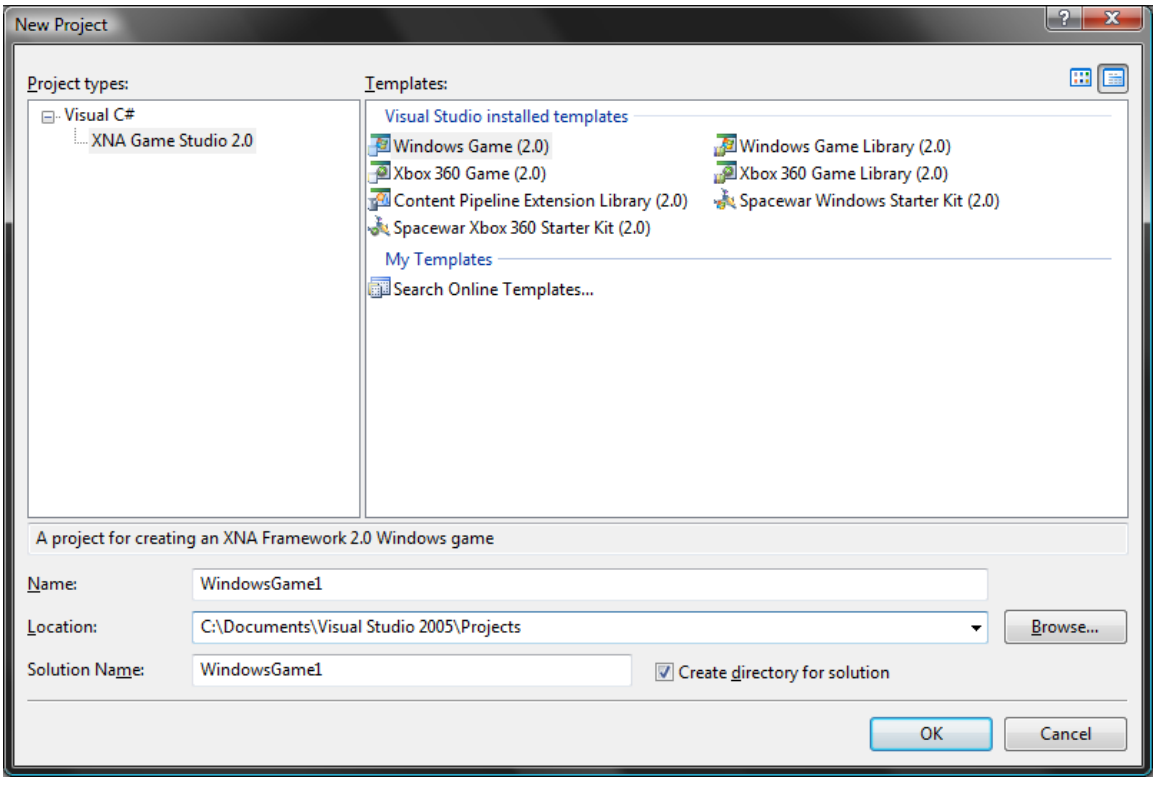
Kuvio 5.1. Vuokaavio. (Cawood & McGee 2007, 24)

5.1 XNA Peliprojektin luonti

Helpoin tapa luoda peliprojekti on käyttää olemassa olevia malleja, jotka tulevat XNA Game Studion mukana. Peliprojektin valintaikkuna näkyy kuviossa 5.2. Peliprojektin luonti tapahtuu seuraavasti:

1. Avataan XNA Game studio, ellei se ole jo auki.
2. Valitaan päävalikosta File ja seuraavaksi valitaan New Project.
3. Valitaan joko Windows tai XBOX 360 Pelimalli.

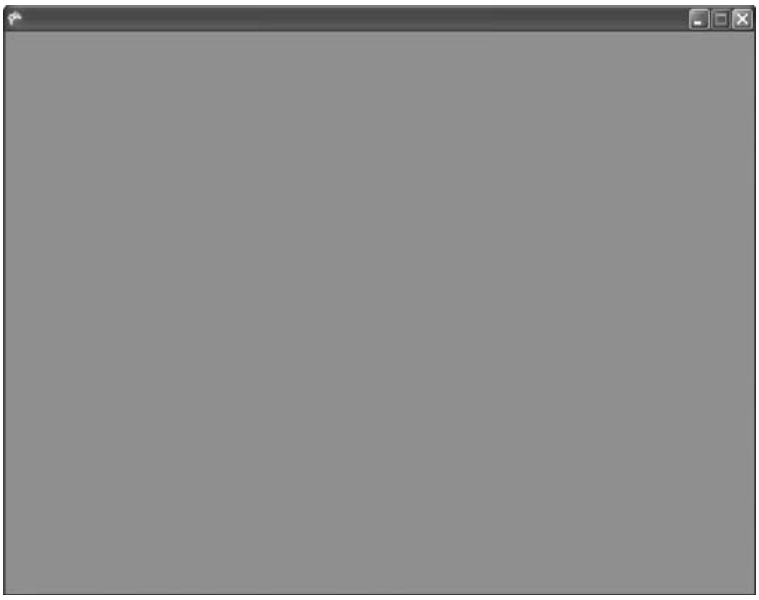
On myös mahdollista luoda XNA peliprojekti tyhjästä, mutta tämä vaatii sen, että käyttäjä lisää itse kaikki XNA:n tarvittavat kirjastoviittaukset peliprojektiin ja samalla käyttäjän tulee lisätä tarvittavat nimiavaruudet peliprojektiin. (Cawood & McGee 2007, 26)



Kuvio 5.2 Peliprojektin valintaikkuna

XNA:ssa peli-ikkunan luonti on todella helppoa. XNA -alusta luo automaattisesti kaikki tarvittavat luokkarakenteet ja metodit, kun käyttäjä luo uuden peliprojektin. Tämän ansiosta käyttäjä pääsee suoraan kehittämään omaa peliä. Peliprojekteja on kaksi eri mallia, toinen on

PC yhteensopiva kun taas toinen on XBOX 360 yhteensopiva. Periaatteessa molemmat projektit luovat samanlaiset koodit alleen. Ainoana eroavaisuutena on, että PC ja XBOX 360 -projekteissa on käytössä eri nimiavaruus. Peruspeli-ikkuna näkyy kuviossa 5.3. (Cawood & McGee 2007, 25)



Kuvio 5.3. Peruspeli-ikkuna (Cawood & McGee 2007, 25)

5.1.1 Pelin alustus

Jokainen XNA peliprojekti vaatii toimiakseen **GraphicsDeviceManager** -objektin, jolla hallitaan näytönohjainta ja sen asetuksia. GraphicsDeviceManager -objektia käytetään primitiivisten elementtien piirrossa. GraphicsDeviceManager -objekti määritellään peliprojektin alussa Game -luokan jäsenmuuttujana ja alustetaan Game -luokan konstruktorissa kuviossa 5.4 esitetyllä tavalla.

```
// Määrittely
GraphicsDeviceManager graphics;

// Alustus
graphics = new GraphicsDeviceManager(this);
```

Kuvio 5.4. GraphicsDeviceManager -objektin määrittely ja alustus (Cawood & McGee 2007, 26)

ContentManager -objektia käytetään sisällön lataamiseen, hallintaan ja vapauttamiseen. Graafinen sisältö ja mediasisältö voidaan ladata tällä objektilla, jos kyseisistä sisällöistä on tehty viittaus peliprojektiin. ContentManager -objekti määritellään peliprojektin alussa Game -luokan jäsenmuuttujana ja alustetaan Game -luokan konstruktorissa kuviossa 5.5 esitetyllä tavalla.

```
//Määrittely
ContentManager content;

//Alustus
content = new ContentManager(Services);
```

Kuvio 5.5 ContentManager objektin määrittely ja alustus (Cawood & McGee 2007, 26)

Initialize -metodi ajetaan kerran pelin käynnistyessä ja tämän metodin sisällä suoritetaan kaikki pelin kannalta tärkeät alustukset, kuten:

- peli-ikkunan asetusten määritys kuten otsikko, peli-ikkunan koko jne.,
- perspektiivin ja näkymän asettaminen, eli se miten pelaaja näkee mahdollisen 3D-maailman pelin alussa,
- image objektien alustus teksturointia varten,
- verteksit, jotka sisältävät mahdollisia väri- ja paikkatietoja tekstuureita ja muita toimintoja varten,
- verteksivarjostimet, alustamalla varjostimien objektit ja lataamalla kehittäjien määräämät varjostinkoodit alustettuihin luokkiin,
- ääniobjektit, alustamalla pelin äänien objektit ja lataamalla kehittäjien määräämät äänitiedostot alustettuihin luokkiin,
- muut peliobjektit, alustamalla kaikki pelin kannalta tärkeät kehittäjien ohjelmoimat luokat.

(Cawood & McGee 2007, 27)

LoadGraphicsContent -metodia käytetään yleensä binäärimedian, kuten kuvien ja mallien lataamiseen. Käyttäjä voi ladata binäärimedian myös Initialize -metodissa, mutta LoadGraphicsContent -metodi mahdollistaa sen, että kaikki binäärimedia ladataan yhden metodin sisällä. Käyttämällä peliprojektimallia tämä metodi lisätään automaattisesti koodiin. (Cawood & McGee 2007, 27)

5.1.2 Pelin piirto ja päivitys

XNA pelin alustuksen jälkeen, käynnistetään jatkuva silmukka, joka ajaa piirto- ja päivitysajoja. Yleensä nämä ajot tapahtuvat vuorotellen eli piirtoajon jälkeen ajetaan päivitysajo jne. Mutta asia ei ole aina näin, välillä voi tulla useita piirtoajoja ennen päivitysajoa ja sama voi tapahtua toisinpäin eli välillä tulee useita päivitysajoja ennen piirtoajoa. Tästä johtuen piirtoajot ja päivitysajot tulisi olla sidottuna peliaikaan, jotta piirrot ja päivitykset olisivat yhtenevät. Kaikkien graafisten objektien piirto tapahtuu Draw -metodissa. Kaikki pelilogiikan päivitykset, erilaisten peliobjektien seuranta ja tapahtumien seuranta tapahtuu Update -metodissa. Molemmat näistä metodeista ajetaan jokaisen ruudunpäivityksen yhteydessä. (Cawood & McGee 2007, 27)

Draw -metodilla hallitaan kaikki pelissä tapahtuva piirto. Draw -metodi sisältää yleensä seuraavat toiminnot:

- Näyttöalueen tyhjentäminen eli täytetään näyttöalue jollakin käyttäjän määräämällä värillä
- Piirretään graafiset objektit näytölle

(Cawood & McGee 2007, 27)

Update -metodissa käsitellään yleensä kaikki peliin liittyvät tapahtumat ja muutokset. Joitakin käsiteltäviä tapahtumia ovat hiiren painallukset ja liike, näppäimistön painallukset, peliohjaimen painallukset ja ajastimet. Update -metodissa suoritetaan myös muita toimintoja, jotka vaativat kokoaikaista seurantaa ja päivittämistä, kuten esimerkiksi animointi ja törmäystarkistus. (Cawood & McGee 2007, 28)

Peliprojektimallia käytettäessä pelin koodiin lisätään automaattisesti UnLoadGraphicsContent -metodi. Tällä metodilla yleensä vapautetaan kaikki ladatut ja alustetut graafiset mediat ohjelman sulkemisen yhteydessä. Tämä metodi ajetaan myös silloin, kun peli sammuu tai kaatuu satunnaisesti. (Cawood & McGee 2007, 28)

5.1.3 Esimerkki XNA peli-ikkunasta

Tässä esimerkissä esitetään C# koodi, jonka XNA Game Studio automaattisesti luo, kun käyttäjä luo peruspeliprojektin käyttäen Windows tai XBOX 360 -projektimallia. Luonnin yhteydessä XNA alusta luo automaattisesti kaksi tiedostoa, joista toinen on Program1.cs. Tämän tiedoston tarkoitus on käynnistää peli. Tiedoston sisältö on esitettynä kuviossa 5.6.

```
using System;

namespace WindowsGame
{
    static class Program
    {
        // Ohjelman käynnistymispiste.
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

Kuvio 5.6. Program1.cs tiedoston sisältö (Cawood & McGee 2007, 28)

Toinen näistä tiedostoista on Game1.cs. Tämä tiedosto sisältää pelin kannalta kaikki tarvittavat koodit kuten alustukset, piirrot, päivitykset jne. Tiedoston sisältö on esitettynä kuviossa 5.7.

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
#endregion
```

```
namespace WindowsGame1
{
     public class Game1 : Microsoft.Xna.Framework.Game
     {
         GraphicsDeviceManager graphics; // Piirron hallintaan

         // lataa, hallitsee ja vapauttaa gfx mediaa
         ContentManager content;

      public Game1()
      {
            // Alustaa graphics ja content objektit
            graphics = new GraphicsDeviceManager(this);
            content = new ContentManager(Services);
      }

      protected override void Initialize()
      {
            // Peli-ikkunan alustus, tähän tulee kaikki tarvittavat
            // alustukset
            base.Initialize();
      }

      // load graphics content
      protected override void LoadGraphicsContent(bool loadAllContent)
      {
            if (loadAllContent)
            { } // lataa hallittu(managed) graafinen sisältö
            else
            { } // lataa ei hallittu (un-managed)graafinen sisältö
      }

      protected override void UnloadGraphicsContent(bool unloadAllCon
      tent)
      {
            // vapauta kaikki graafinen sisältö
            if (unloadAllContent == true)
            {
                  content.Unload();
            }
      }



      protected override void Update(GameTime gameTime)
      {
            // Animaatiot, törmäystentarkistus, tapahtumien hallinta
            // Mahdollistaa pelin sammutuksen, nappia painetessa
            if(GamePad.GetState(PlayerIndex.One).Buttons.Back ==
            ButtonState.Pressed)
                  this.Exit();

            base.Update(gameTime);
      }

      protected override void Draw(GameTime gameTime)
      {
            // Piirrä ikkunaan
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
```

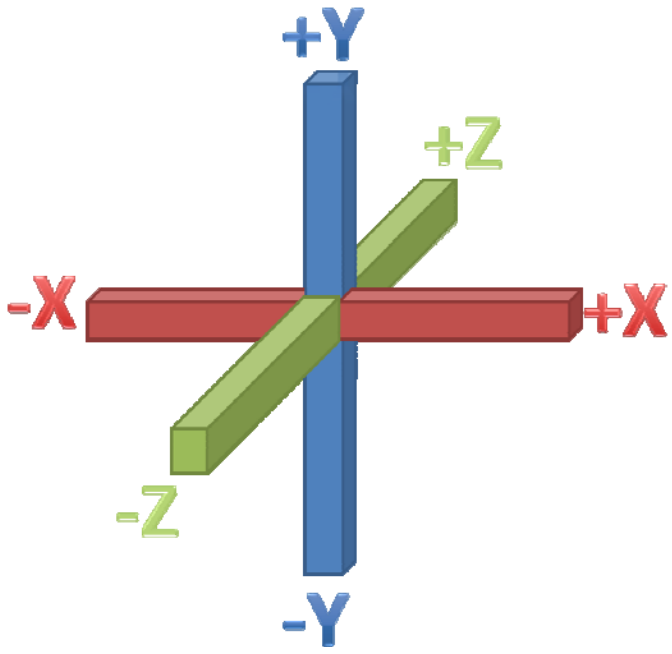
```
            // kutsu varjostin koodia täältä
            base.Draw(gameTime);
        }
    }
}
```

Kuvio 5.7. Game1.cs tiedoston sisältö (Cawood & McGee 2007, 28-30.)

Tässä on kaikki koodi mitä tarvitaan kuviossa 5.3 näkyvän peli-ikkunan luomiseen, joten XNA helpottaa peruspeli-ikkunan luontia. Käytettäessä uuden peliprojektin luomisessa peli-projektimallia, XNA Game Studio generoi em. koodit automaattisesti. (Cawood & McGee 2007, 30)

5.2 Perusmuotojen piirto

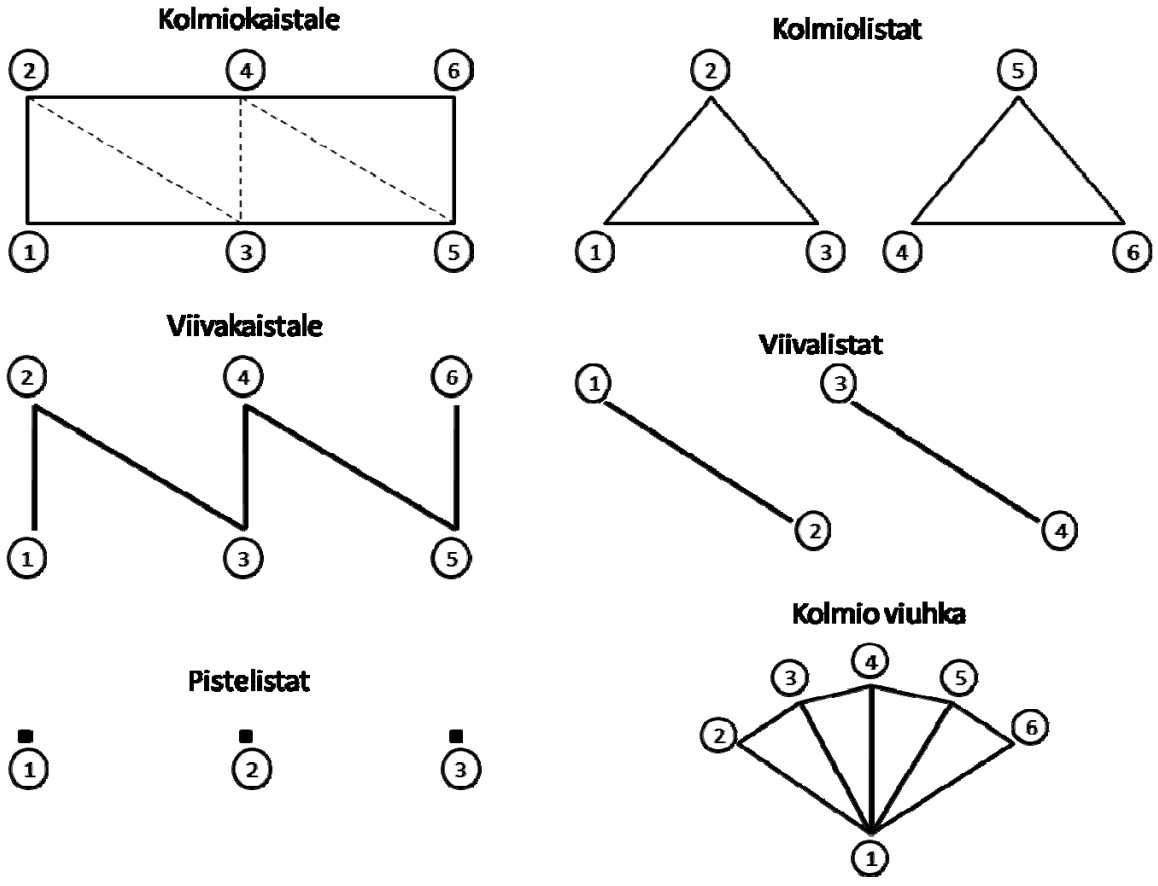
Perusgrafiikkaa muodostetaan ja piirretään pisteillä, viivoilla ja kolmiolla. Näitä perusele-menttejä kutsutaan usein primitiiviobjekteiksi. Primitiiviobjektit piirretään 3D-maailmaan käyttäen karteesista koordinaatistoa, jossa paikkatiedot ovat esitettyinä X, Y ja Z akseleilla (Kuvio 5.8). (Cawood & McGee 2007, 31)



Kuvio 5.8 Karteesinen koordinaatisto.

Kaikki mallit ja muodot koostuvat lukemattomista pisteistä, viivoista ja kolmioista. Staattinen 3D-malli on periaatteessa vain tiedosto, johon on tallennettuna verteksitietoa sisältäen X, Y ja Z-akseleiden paikkatiedot, väritiedot ja joitakin muita tietoja. Verteksit eli kulmapisteet voidaan renderöidä piirtämällä ainoastaan verteksien pisteet tai piirtämällä verteksejä yhdistävät viivat tai jopa piirtämällä pinnat, jotka muodostuvat verteksien muodostamista viivoilla yhdistetyistä kolmioista. (Cawood & McGee 2007, 31)

Monimutkaiset muodot muodostetaan primitiiviobjekteilla, joilla määritetään miten verteksit näytetään. Verteksitieto voidaan esittää pisteillä, viivoilla tai kiinteillä kolmioilla. Kuviossa 5.9 on esitettynä nämä eri vaihtoehdot. (Cawood & McGee 2007, 31)



Kuvio 5.9. Primitiiviobjektien esitysmuodot. (Cawood & McGee 2007, 31)

XNA alusta tarjoaa helpon syntaksin muotojen piirtämiseen primitiiviobjekteista. (Cawood & McGee 2007, 32)

Kuviossa 5.9 on esitettynä kuuden yleisimmän primitiiviobjektin käyttömahdollisuudet. Kuten kuviosta näkyy, viivoja ja kolmioita voidaan piirtää kaistaleina, listoina ja viuhkoina. Listoja käytetään, kun piirretään erillisiä pisteitä, viivoja tai kolmioita. Kaistaleita käytetään yhtenäisten muotojen piirrossa ja ne ovat siten listoja tehokkaampia yhtenäisten muotojen piirrossa. Viuhkoja käytetään yleensä pyöreiden muotojen piirrossa. Kaistaleet ovat muistinkannalta paljon tehokkaampia kuin listat. Tästä johtuen kaistaleiden piirto on nopeampaa verrattuna listoihin. Kolmiokaistaletta käytettäessä, lisäämällä yhden verteksin saadaan aikaiseksi yksi uusi kolmio ja siten kaistale käytännössä puolittaa muistinkulutuksen. Seuraavilla kaavoilla voidaan havainnollistaa tämä tehokkuus:

$$\text{Verteksimäärä listassa} = N_{kolmiot} * 3 \text{ verteksiä}$$

$$\text{Verteksimäärä kaistaleessa} = N_{kolmiot} + 2 \text{ verteksiä}$$

Sama pätee myös viivoihin:

$$\text{Viivamäärä listassa} = N_{viivat} * 2 \text{ verteksiä}$$

$$\text{Viivamäärä kaistaleessa} = N_{viivat} + 1 \text{ verteksi}$$

(Cawood & McGee 2007, 32)

Verteksiobjektit sisältävät verteksitietoa, joka voi koostua X, Y ja Z -paikkatiedoista, kuvakoordinaateista, normaalivektorista ja väristä. XNA alusta tarjoaa neljä erilaista esimääriteltyä verteksiformaattia, jotka on esitetty taulukossa 5.1. (Cawood & McGee 2007, 32)

Verteksitietojen formaatit	
VertexPositionColor	Säilyttää X-, Y- ja Z-paikkatiedot ja värin.
VertexPositionTexture	Säilyttää X-, Y- ja Z-paikkatiedot ja kuvakoordinaatit.
VertexPositionNormal	Säilyttää X-, Y- ja Z-paikkatiedot ja normaalivektorin.
VertexPositionNormalTexture	Säilyttää X-, Y- ja Z-paikkatiedot, normaalivektorin ja kuvakoordinaatit.

Taulukko 5.1. Verteksiformaatit. (Cawood & McGee 2007, 33)

VertexDeclaration objekti pitää sisällään verteksiformaatin, jonka perusteella näytönohjain osaa piirtää verteksit oikein näytölle. Ennen verteksien piirtoa täytyy alustaa niitä vastaava VertexDeclaration objekti, jotta näytönohjain osaa poimia verteksitiedot oikein vertekseistä piirron yhteydessä. Kuviossa 5.10 on esitetty miten määritellään ja alustetaan VertexDeclaration objekti.

```
//Luodaan vertex declaration objekti
VertexDeclaration vertexDeclaration = new VertexDeclaration(
gfx.GraphicsDevice, VertexPositionColor.VertexElements);

//Näytönohjaimen vertexdeclaration asettaminen
gfx.GraphicsDevice.VertexDeclaration = vertexDeclaration;
```

Kuvio 5.10. VertexDeclaration objektin määrittely ja alustus (Cawood & McGee 2007, 33)

Seuraavat viisi asetusta täytyy tehdä piirrettäessä objekti primitiivityypeillä:

1. Määritellään käytettävä verteksityyppi
2. Määritellään käytettävä primitiivityyppi
3. Alustetaan ja täytetään verteksitaulukko, joka voi sisältää X, Y ja Z -arvot, värin, tekstuurin ja normaalivektorin. Näiden arvojen perusteella objekti piirretään.
4. Määritellään alkuelementti verteksitaulukosta, josta piirto aloitetaan.
5. Määritellään piirtävien elementtien määrä verteksitaulukosta.

(Cawood & McGee 2007, 33)

Edellä mainitut asetukset syötetään DrawUserPrimitives metodille kuviossa 5.11 esitetyllä tavalla.

```
gfx.GraphicsDevice.DrawUserPrimitives<struct customVertex>(
enum PrimitiveType, struct customVertex vertices,
int startingVertex,int primitiveCount);
```

Kuvio 5.11. DrawUserPrimitives metodin käyttö (Cawood & McGee 2007, 33)

Muotojen piirtämiseen käytetään siihen sopivia primitiivityyppejä. Käyttäjän pitää huomioida pari asiaa ennen kuin perehdytään tarkemmin piirtoesimerkkeihin. Verteksityypitykselle pitää määrittää VertexDeclaration objekti peliluokan alkuun. Piirron yhteydessä tällä objektilla

määritellään mitä verteksejä piirrettävä muoto käyttää. VertexDeclaration objekti määritellään kuviossa 5.10 esitetyllä tavalla.

5.2.1 Kolmiokaistaleen piirto

Kolmiokaistaleen piirto aloitetaan määrittelemällä verteksitaulukko, joka pitää sisällään piirrettävät verteksit. Verteksitaulukko määritellään Game luokan jäsenmuuttujaksi ja verteksitaulukon tyyppi pitää olla VertexPositionColor, koska piirrettävät verteksit sisältävät ainoastaan väri- ja paikkatiedot. Neljän kulmapisteen verteksitaulukko määritellään ja alustetaan kuviossa 5.12 esitetyllä tavalla.

```
//Määrittely, Game luokan alkuun
private VertexPositionColor[] mVtTriStrip = new VertexPositionColor[4];

//Verteksitaulukon alustusmetodi, Ajetaan Initialize-metodissa
private void init_tri_strip()
{

 mVtTriStrip[0]=new VertexPositionColor(new Vector3(-1.5f,0.0f,3.0f),
 Color.Orange);

 mVtTriStrip[1]=new VertexPositionColor(new Vector3(-1.0f,.5f, 3.0f),
 Color.Orange);

 mVtTriStrip[2]=new VertexPositionColor(new Vector3(-0.5f, 0.0f,3.0f),
 Color.Orange);

 mVtTriStrip[3]=new VertexPositionColor(new Vector3( 0.0f, 0.5f,3.0f),
 Color.Orange);

}


//Initialize-metodissa suoritettava metodikutsu
init_tri_strip();
```

Kuvio 5.12. Verteksitaulukon määrittely ja alustus(Cawood & McGee 2007, 35- 36)

Seuraavaksi tarvitaan metodi, joka piirtää kyseiset verteksit näytölle. Primitiiviobjektin piirto tapahtuu yleensä noudattaen seuraavia vaiheita:

1. Määritellään muunnosmatriisit objektin muokkausta varten.

2. Alustetaan muunnosmatriisit.

3. Rakennetaan kumulatiivinen muunnos kertomalla matriisit keskenään.

4. Lähetetään syötetietoina kumulatiivinen muunnos varjostimelle.

5. Valitaan verteksityyppi, primitiivityyppi, piirrettävät verteksit ja piirretään objekti.

(Cawood & McGee 2007, 36)

Piirtometodi DrawTriangleStrip on esitetty kuviossa 5.13.

```
void DrawTriangleStrip()
{
      // 1: määritellään matriisit
      Matrix matIdentity;

      // 2: alustetaan matriisit
      matIdentity = Matrix.Identity;

      // Aloita aina yksikkömatriisilla

      // 3: rakennetaan kumulatiivinen matriisi käyttäen
      //I.S.R.O.T. sekvenssiä

      // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja siirto
      // (ENG) identity, scale, rotate, orbit(translate & rotate),
      //translate
      mMatWorld = matIdentity;

      // 4: lähetetään syötetietoina kumulatiivinen
      //transformaatio matriisi varjostimelle
      worldViewProjParam.SetValue(mMatWorld * mMatView * mMat - Proj);

      // Hyväksytetään muutokset varjostimella
      mfx.CommitChanges();

      // 5: piirrä objekti – valitse verteksityyppi,
      //primitiivityyppi, piirrettävien verteksien määrä
      gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
      gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
      PrimitiveType.TriangleStrip, mVtTriStrip, 0, 2);

 }
```

Kuvio 5.13. Kolmiokaistaleen piirtometodi (Cawood & McGee 2007, 36-37.)

DrawTriangleStrip -metodia pitää kutsua peliluokan Draw -metodin sisältä, jonka jälkeen peli piirtää verteksit näytölle. DrawTriangleStrip -metodia kutsutaan seuraavalla tavalla Draw -metodin sisällä:

```
DrawTriangleStrip();
```

(Cawood & McGee 2007, 37)

5.2.2 Kolmiolistan piirto

Kolmiolistan piirto aloitetaan määrittelemällä verteksitaulukko, joka pitää sisällään piirrettävät verteksit. Verteksitaulukko määritellään Game luokan jäsenmuuttujaksi ja verteksitaulukon tyyppi pitää olla VertexPositionColor. Kuuden kulmapisteen verteksitaulukko määritellään ja alustetaan kuviossa 5.14 esitetyllä tavalla.

```
//Määrittely Game luokan alkuun
private VertexPositionColor[] mVtTriList = new VertexPositionColor[6];

//Verteksitaulukon alustusmetodi, Ajetaan Initialize-metodissa
private void init_tri_list()
{
   mVtTriList[0] = new VertexPositionColor(new Vector3(0.5f,0.0f,3.0f),
   Color.LightGray);

   mVtTriList[1] = new VertexPositionColor(new Vector3(0.7f, 0.5f,
   3.0f), Color.LightGray);

   mVtTriList[2] = new VertexPositionColor(new Vector3(0.9f, 0.0f,
   3.0f), Color.LightGray);

   mVtTriList[3] = new VertexPositionColor(new Vector3(1.1f, 0.0f,
   3.0f), Color.LightGray);

   mVtTriList[4] = new VertexPositionColor(new Vector3(1.3f, 0.5f,
   3.0f), Color.LightGray);

   mVtTriList[5] = new VertexPositionColor(new Vector3(1.5f, 0.0f,
   3.0f), Color.LightGray);
}

//Initialize-metodissa suoritettava metodikutsu
init_tri_list();
```

Kuvio 5.14. Verteksitaulukon määrittely ja alustus (Cawood & McGee 2007, 38)

Seuraavaksi tarvitaan metodi, joka piirtää kyseiset verteksit näytölle. Kolmiolistan piirtometodi DrawTriangleList on esitetty kuviossa 5.15.

```
void DrawTriangleList()
{
      // 1: määritellään matriisit
      Matrix matIdentity;

      // 2: alustetaan matriisit
      matIdentity = Matrix.Identity;

      // Aloita aina yksikkömatriisilla

      // 3: rakennetaan kumulatiivinen matriisi käyttäen
      //I.S.R.O.T. sekvenssiä
      // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja siirto
      // (ENG) identity, scale, rotate, orbit(translate & rotate),
      // translate
      mMatWorld = matIdentity;

      // 4: lähetetään syötetietoina kumulatiivinen
      //transformaatio matriisi varjostimelle
      worldViewProjParam.SetValue(mMatWorld * mMatView * mMatProj);

      // Hyväksytetään muutokset varjostimella
      mfx.CommitChanges();

      // 5: piirrä objekti – valitse verteksityyppi,
      //primitiivityyppi, piirrettävien verteksien määrä
      gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
      gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
      PrimitiveType.TriangleList, mVtTriList, 0, 2);

}
```

Kuvio 5.15. Kolmiolistan piirtometodi (Cawood & McGee 2007, 38)

DrawTriangleList -metodia pitää kutsua peliluokan Draw -metodin sisältä, jonka jälkeen peli piirtää verteksit näytölle. DrawTriangleList -metodia kutsutaan seuraavalla tavalla Draw -metodin sisällä:

DrawTriangleList();

(Cawood & McGee 2007, 38)

5.2.3 Viivakaistaleen piirto

Viivakaistaleen piirto aloitetaan määrittelemällä verteksitaulukko, joka pitää sisällään piirrettävät verteksit. Verteksitaulukko määritellään Game luokan jäsenmuuttujaksi ja verteksitaulukon tyyppi pitää olla VertexPositionColor. Kolmen kulmapisteen verteksitaulukko määritellään ja alustetaan kuviossa 5.16 esitetyllä tavalla.

```
//Määrittely Game luokan alkuun
private VertexPositionColor[] mVtLineStrip =
new VertexPositionColor[3];

//Verteksitaulukon alustusmetodi, Ajetaan Initialize-metodissa
private void init_line_strip()
{
      mVtLineStrip[0] = new VertexPositionColor(new Vector3(1.3f, 0.8f,
      3.0f), Color.Gray);

      mVtLineStrip[1] = new VertexPositionColor(new Vector3(1.0f,
      0.13f, 3.0f), Color.Gray);

      mVtLineStrip[2] = new VertexPositionColor(new Vector3(0.7f, 0.8f,
      3.0f),Color.Gray);
}

//Initialize-metodissa suoritettava metodikutsu
init_line_strip();
```

Kuvio 5.16. Verteksitaulukon määrittely ja alustus (Cawood & McGee 2007, 38 - 39)

Seuraavaksi tarvitaan metodi, joka piirtää kyseiset verteksit näytölle. Viivakaistaleen piirtometodi DrawLineStrip on esitetty kuviossa 5.17.

```
void DrawLineStrip()
{
      // 1: määritellään matriisit
      Matrix matIdentity;

      // 2: alustetaan matriisit
      matIdentity = Matrix.Identity;

      // Aloita aina yksikkömatriisilla

      // 3: rakennetaan kumulatiivinen matriisi käyttäen
      //I.S.R.O.T. sekvenssiä
      // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja siirto
      // (ENG) identity, scale, rotate, orbit(translate & rotate),
      // translate
```

```
        mMatWorld = matIdentity;

        // 4: lähetetään syötetietoina kumulatiivinen
        // transformaatio matriisi varjostimelle
        worldViewProjParam.SetValue(mMatWorld * mMatView * mMatProj);

        // Hyväksytetään muutokset varjostimella
        mfx.CommitChanges();

        // 5: piirrä objekti – valitse verteksityyppi,
        //primitiivityyppi, piirrettävien verteksien määrä
        gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
        gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
        PrimitiveType.LineStrip, mVtLineStrip, 0, 2);

}
```

Kuvio 5.17. Viivakaistaleen piirtometodi (Cawood & McGee 2007, 39)

DrawLineStrip -metodia pitää kutsua peliluokan Draw -metodin sisältä, jonka jälkeen peli piirtää verteksit näytölle. DrawLineStrip -metodia kutsutaan seuraavalla tavalla Draw -metodin sisällä:

DrawLineStrip();

(Cawood & McGee 2007, 39)

5.2.4 Viivalistan piirto

Viivalistan piirto aloitetaan määrittelemällä verteksitaulukko, joka pitää sisällään piirrettävät verteksit. Verteksitaulukko määritellään Game luokan jäsenmuuttujaksi ja verteksitaulukon tyyppi pitää olla VertexPositionColor. Neljän kulmapisteen verteksitaulukko määritellään ja alustetaan kuviossa 5.18 esitetyllä tavalla.

```
//Määrittely Game luokan alkuun
private VertexPositionColor[] mVtLineList = new VertexPositionColor[4];

//Verteksitaulukon alustusmetodi, Ajetaan Initialize-metodissa
private void init_line_list()
{
    mVtLineList[0] = new VertexPositionColor(new Vector3(0.0f, 0.7f,
    3.0f), Color.Black);

    mVtLineList[1] = new VertexPositionColor(new Vector3(-1.0f, 0.7f,
    3.0f), Color.Black);
```

```
    mVtLineList[2] = new VertexPositionColor(new Vector3(0.0f, 0.8f,
    3.0f),Color.Black);

    mVtLineList[3] = new VertexPositionColor(new Vector3(-1.0f, 0.8f,
    3.0f), Color.Black);
}

//Initialize-metodissa suoritettava metodikutsu
init_line_list();
```

Kuvio 5.18. Verteksitaulukon määritys ja alustus (Cawood & McGee 2007, 40)

Seuraavaksi tarvitaan metodi, joka piirtää kyseiset verteksit näytölle. Kolmiolistan piirtometodi DrawLineList on esitetty kuviossa 5.19.

```
void DrawLineList()
{
      // 1: määritellään matriisit
      Matrix matIdentity;

      // 2: alustetaan matriisit
      matIdentity = Matrix.Identity;

      // Aloita aina yksikkömatriisi

      // 3: rakennetaan kumulatiivinen matriisi käyttäen
      //I.S.R.O.T. sekvenssiä

      // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja siirto
      // (ENG) identity, scale, rotate, orbit(translate & rotate),
      // translate
      mMatWorld = matIdentity;

      // 4: lähetetään syötetietoina kumulatiivinen
      //transformaatio matriisi varjostimelle
      worldViewProjParam.SetValue(mMatWorld * mMatView * mMatProj);

      // Hyväksytetään muutokset varjostimella
      mfx.CommitChanges();

      // 5: piirrä objekti – valitse verteksityyppi,
      //primitiivityyppi, piirrettävien verteksien määrä
      gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
      gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
      PrimitiveType.LineList, mVtLineList, 0, 2);

}
```

Kuvio 5.19. Kolmiolistan piirtometodi (Cawood & McGee 2007, 40)

DrawLineList -metodia pitää kutsua peliluokan Draw -metodin sisältä, jonka jälkeen peli piirtää verteksit näytölle. DrawLineList -metodia kutsutaan seuraavalla tavalla Draw -metodin sisällä:

DrawLineList();

(Cawood & McGee 2007, 40)

5.2.5 Pistelistan piirto

Pistelistan piirto aloitetaan määrittelemällä verteksitaulukko, joka pitää sisällään piirrettävät verteksit. Verteksitaulukko määritellään Game luokan jäsenmuuttujaksi ja verteksitaulukon tyyppi pitää olla VertexPositionColor. Kahden kulmapisteen verteksitaulukko määritellään ja alustetaan kuviossa 5.20 esitetyllä tavalla.

```
//Määrittely Game luokan alkuun
private VertexPositionColor[] mVtPointList =
      new VertexPositionColor[2];

//Verteksitaulukon alustusmetodi, Ajetaan Initialize-metodissa
private void init_pointList()
{
      mVtPointList[0]=new VertexPositionColor(new Vector3(0.25f, 0.8f,
      3.0f), Color.Black);

      mVtPointList[1]=new VertexPositionColor(new Vector3(0.25f, 0.0f,
      3.0f), Color.Black);
}

//Initialize-metodissa suoritettava metodikutsu
init_pointList();
```

Kuvio 5.20. Verteksitaulukon määrittely ja alustus (Cawood & McGee 2007, 41)

Seuraavaksi tarvitaan metodi, joka piirtää kyseiset verteksit näytölle. Pistelistan piirtometodi DrawPointList on esitetty kuviossa 5.21.

```
void DrawPointList()
{
      // 1: määritellään matriisit
      Matrix matIdentity;

      // 2: alustetaan matriisit
      matIdentity = Matrix.Identity;
```

```
        // 3: rakennetaan kumulatiivinen matriisi käyttäen
        //I.S.R.O.T. sek-venssiä
        // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja siirto
        // (ENG) identity, scale, rotate, orbit(translate & rotate),
        //translate
        mMatWorld = matIdentity;

        // 4: lähetetään syötetietoina kumulatiivinen
        //transformaatio matriisi varjostimelle
        worldViewProjParam.SetValue(mMatWorld * mMatView * mMatProj);

       // Hyväksytetään muutokset varjostimella
        mfx.CommitChanges();

        // 5: piirrä objekti – valitse verteksityyppi,
        //primitiivityyppi, piirrettävien verteksien määrä
        gfx.GraphicsDevice.VertexDeclaration = mVertexDeclaration;
        gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
        PrimitiveType.PointList, mVtPointList, 0, 2);

}
```

Kuvio 5.21. Pistelistan piirtometodi (Cawood & McGee 2007, 41)

DrawPointList -metodia pitää kutsua peliluokan Draw -metodin sisältä, jonka jälkeen peli piirtää verteksit näytölle. DrawPointList -metodia kutsutaan seuraavalla tavalla Draw -metodin sisällä:

```
DrawPointList();
```

(Cawood & McGee 2007, 41)

5.3 3D Mallit

Kun peliin tahdotaan realistisempia ja monimutkaisempia malleja, käsin koodatut ja alkeelliset mallit eivät riitä ja monimutkaisten mallien luonti näin tuottaa liikaa työtä. 3D mallinnusohjelman avulla mutkikkaiden mallien luonti on helpompaa ja nopeampaa. Lisäksi malleista saadaan vähemmällä vaivalla kiinnostavampia ja realistisempia. Valmiiksi luodut 3D -mallit voidaan lisätä peliin ja niitä voidaan kontrolloida koodissa. XNA tukee tällä hetkellä kahta eri 3D mallien tiedostomuotoa, jotka ovat Microsoftin DirectX -ohjelmointirajapinnan x-tiedostomuoto ja Autodeskin fbx-tiedostomuoto. Myös muiden muotojen käyttö on mahdollista, mutta tällöin tarvitaan erillinen mallien lataaja. (Cawood & McGee 2007, 164 ; Autodesk, 2008)

5.3.1 Mallien lataus XNA:ssa

3D-mallien hallintaan, käsittelyyn ja piirtoon XNA tarjoaa Model -luokan. Luokan avulla mallien lataus, muuntaminen ja piirto ovat helppoa. Luokka käyttää luurankomaista hierarkiaa mallien verteksien säilöntään ja piirtoon. Luuranko koostuu luista, joilla jokaisella on transformaatio suhteessa luurankoon. Jokaisella luulla on myös tiedot siihen liittyvistä vertekseistä. Verteksit taas sisältävät paikkatiedon, normaalit ja teksturointi-informaation, jotka liikkuvat luiden mukana, kun luihin kohdistuu transformaatio. Model -luokka sisältää matriisitaulun, jossa jokaiselle luulle on määritetty oma transformaatiomatriisi, joka määrittää luun sijainnin luurangossa. Luuranko voi olla osa ladattavaa 3D -mallia kuten kuvassa (Kuvio 5.22). (Cawood & McGee 2007, 176)

Kuvio 5.22. 3D malli ja luut (Project BlueStreak, 2006)

Mallit lisätään peliprojektiin lisäämällä niihin ensin viittaus. Viittaus lisätään Solution Explorerista. Viittaus tulee lisätä kaikille malleille, joita peliprojektissa käytetään. Mallien lataus Model luokkaan tapahtuu Content Pipelinen Load -metodia apuna käyttäen (Kuvio 5.23). (Cawood & McGee 2007, 176 ; Nitschke 2007, 66-68.)

```
Model model1 = content.Load<Model>(".\\Kansio\\mallin_nimi");
```

Kuvio 5.23. 3D mallin lataus

5.3.2 Mallien piirto

Malli voi sisältää useamman mesh -objektin ja jokaisella mesh -objektilla on omat verteksit, tekstuurit sekä normaalit. Mallin piirrossa eli renderöinnissä tämä tulee ottaa huomioon. Tämä onnistuu käymällä läpi foreach -silmukassa kaikki model -objektista löytyvät ModelMesh -objektit. Näin kaikille mesh -objekteille saadaan transformaatio maailma koordinaatiistoon nähden ja valaistus, jos sille on tarvetta sekä kaikki mallin mesh -objektit tulevat myös varmasti piirretyksi. Mallin piirto on esitetty kuviossa 5.24.

```
foreach (ModelMesh mesh in model.Meshes)
{
      foreach (BasicEffect effect in mesh.Effects)
      {
      //Tarvittavat matriisit julkisia ja alustetaan muualla
      //(view, projection ja world)
      effect.View = _matView;
      effect.Projection = _matProjection;
      effect.World = _matWorld;
      effect.EnableDefaultLighting();
      mesh.Draw();
      }
}
```

Kuvio 5.24. Mallien piirto

Tämän silmukan on sisällettävä vielä toinen silmukka, jotta malli saadaan piirrettyä oikein. Sisemmässä silmukassa käydään läpi kaikki mesh -objektin BasicEffect -objektit ja sijoitetaan niihin tarvittavat muunnokset ja muut asetukset. BasicEffect -objekteihin täytyy asettaa View ja Projection matriisit, jotta malli tulee piirretyksi oikein suhteessa kameraan. Jotta kaikkiin mallin mesh -objekteihin saadaan asetettua muunnos, täytyy jokaisen mesh -objektin luu kertoa joko maailmamatriisilla tai muulla yhteisesti vaikuttavalla matriisilla. Mesh -objektiin voidaan asettaa perusvalaistus EnableDefaultLighting -metodilla tai oma muokattu valaistus. Lopuksi mesh -objekti piirretään kutsumalla Draw -metodia. (Cawood & McGee 2007, 177)

5.3.3 Animointi

Yksinkertaiset animaatiot voidaan tehdä suoraan koodissa. Esimerkiksi tyylimyllyn lapojen pyöritys on kohtuullisen helppo toteuttaa. Tuulimylly lavat ja runko on talletettu eri malleihin ja ladataan erillisiin Model -luokkiin. Molemmille malleille tarvitaan myös matriisit transformaatiota varten. Näiden alustuksien jälkeen voidaan ladata mallit (Kuvio 5.25).

```
Model mModBase; Model mModFan;
Matrix[] matFan; Matrix[] matBase;

mModBase = content.Load<Model>(".\\Models\\base");
matBase = new Matrix[mModBase.Bones.Count];
mModBase.CopyAbsoluteBoneTransformsTo(matBase);
mModFan = content.Load<Model>(".\\Models\\fan");
matFan = new Matrix[mModFan.Bones.Count];
mModFan.CopyAbsoluteBoneTransformsTo(matFan);
```

Kuvio 5.25. Mallien lataus ja matriisien alustus

Mallien lisäksi määritellään kaksi vakiomuuttujaa (Kuvio 5.26). Näillä muuttujilla määrätään, kumpi malleista piirretään. Lapojen pyörähdyksen määrää varten luodaan liukulukumuuttuja, jotta pyörähdyksestä saadaan sulava ja jatkuva.

```
const int WINDMILL_BASE = 0;
const int WINDMILL_FAN = 1;
private float mfFanRotation = 0.0f; //Tuulimyllyn lavan kulma
```

Kuvio 5.26. Muuttujat

Mallien piirto on varsin yksinkertaista ja suoraviivaista (Kuvio 5.27). Ensin luodaan ja alustetaan tarvittavat matriisit. Seuraavaksi lasketaan lapojen pyöräytyksen uusi arvo, mutta vain jos piirrettävä malli on lavat. Tämän jälkeen lasketaan maailma matriisi I.S.R.O.T. järjestyksen mukaisesti. Ennen mallien varsinaista piirtoa käydään läpi mallien BasicEffect -objektit ja tehdään tarvittavat asetukset kuten maailma matriisi, valaistus jne. Lopuksi piirretään itse malli Draw -metodia kutsuen.

```
void draw_windmill(Model model, int iModel, GameTime gameTime)
{
      foreach (ModelMesh mesh in model.Meshes)
      {
      Matrix matWorld,matIdent,matScale,matRotY,matRotZ,matTransl;
      // Matriisien alustus
      matIdent = Matrix.Identity;
      matScale = Matrix.CreateScale(0.1f, 0.1f, 0.1f);
      matTransl = Matrix.CreateTranslation(0.0f, 0.0f, 1.5f);
      matRotY = Matrix.CreateRotationY((float)Math.PI);
      matRotZ = Matrix.CreateRotationZ(0.0f);

      if (iModel == WINDMILL_FAN)
      {
            // Lasketaan pyöräytys framejen välille laitteistosta
            //riippumattomalle nopeudelle
            mfFanRotation += gameTime.ElapsedRealTime.Ticks/6000000.0f;

            // Ylivuodon esto
            mfFanRotation=mfFanRotation%(2.0f*(float)Math.PI);
            matRotZ = Matrix.CreateRotationZ(mfFanRotation);
      }
```

```
        // Maailma matriisin luonti I.S.R.O.T.:n mukaisesti
        // identity, scale, rotate, orbit(translate&rotate), translate
        matWorld = matIdent*matScale*matRotZ*matRotY * matTransl;

        foreach (BasicEffect effect in mesh.Effects)
        {
        if(iModel == WINDMILL_BASE)
              effect.World = matBase[mesh.ParentBone.Index]* matWorld;

        if(iModel == WINDMILL_FAN)
              effect.World = matFan[mesh.ParentBone.Index] * matWorld;

        effect.View = mMatView;
        effect.Projection = mMatProj;

        // Valaistuksen asetus
        effect.EnableDefaultLighting();
        effect.CommitChanges();
}

// Objektin piirto
mesh.Draw();
}
```

Kuvio 5.27. Tuulimyllyn piirto ja animointi -metodi

Jotta molemmat mallit saadaan piirrettyä kutsutaan pelin Draw -metodissa aikaisemmin luotua draw_windmill -metodia sekä rungolle että lavoille (Kuvio 5.28). Parametreinä annetaan piirrettävä malli, mallin tunniste sekä peliaika. (Cawood & McGee 2007, 178-181.)

```
draw_windmill(mModBase, WINDMILL_BASE, gameTime);
draw_windmill(mModFan, WINDMILL_FAN, gameTime);
```

Kuvio 5.28. Piirtojen kutsuminen

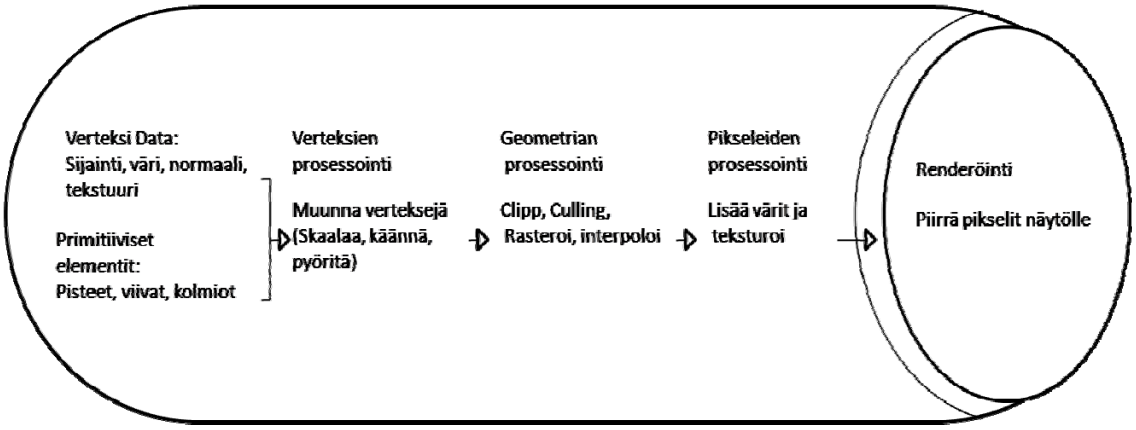
5.4 Varjostimet

XNA käyttää varjostimia (Shader) ja näin ollen varjostinpohjaista renderöintiä muuntaessa verteksidataa pikselidataksi. Tällä tavalla saadaan grafiikan piirto tehokkaammaksi, koska varjostimet siirtävät laskentataakkaa prosessorilta näytönohjaimelle. 3D -grafiikan piirto XNA koodista tapahtuu aina varjostimien avulla riippumatta siitä, että onko ohjelmassa käytetty omia tai XNA:n Basic Effect varjostimia. Varjostimien avulla voidaan määrittää miten verteksit näytetään näytöllä. Varjostimien avulla voidaan muokata kaikkia verteksien ominaisuuksia esim. väri, sijainti ja tekstuuri. Varjostimien tuoma mahdollisuus muokata verteksejä mahdollistaa varjostimien käytön valon, läpinäkyvyyden ja multiteksturoinnin toteuttamisessa. Joidenkin efektien saavuttamiseksi kuten esimerkiksi tulen toteuttamiseen tarkoitetut spri-

tet, multiteksturoinnin ja erikoisvalaistuksen joutuu tekemään kirjoittamalla omat varjostinkoodit. (Cawood & McGee 2007, 44)

5.4.1 Graphics Pipeline

Graphics Pipeline käsittää prosessin, jossa muutetaan syötetyt verteksit ja primitiiviset elementit pikseleiksi. Verteksi- ja pikselivarjostimet ovat avainasemassa kyseisessä prosessissa. Verteksivarjostimien avulla muokataan syötettyjä verteksejä siten, että katsojalle näkymättömät verteksit poistetaan piirrosta. Tätä toimintoa kutsutaan termillä Culling. Rasteroinnilla ja interpoloinnilla muutetaan jäljelle jäävät verteksit kuvaksi. Tekstuurit ja värit lisätään kuvaan pikselivarjostimella ennen kuin kuva näytetään näytöllä. Kuviossa 5.29 näkyy Graphics pipelinen toimintaperiaate. (Cawood & McGee 2007, 44)



Kuvio 5.29. Graphics Pipeline toimintaperiaate. (Cawood & McGee 2007, 44)

5.4.2 Verteksi- ja Pikselivarjostimet

Varjostimien avulla voidaan hallita miten grafiikkaa prosessoidaan. Yleensä joudutaan kirjoittamaan omaa varjostinkoodia. Kuviossa 5.30 esitetty esimerkki verteksivarjostimesta on hyvin yksinkertainen. Se kykenee ainoastaan vastaanottamaan verteksien väri- ja paikkatietoja. Vastaanotettuaan kyseisen tiedon varjostin palauttaa paikkatiedot Graphics Pipelineen. Verteksivarjostimen palauttama tieto interpoloidaan ennen kuin se saavuttaa pikselivarjostimen.

Pikselivarjostin voi tarvittaessa muokata saamansa tietoa. Pikselivarjostin piirtää pikselit näytölle yksitellen saaduista väritiedoista. (Cawood & McGee 2007, 45)

```
float4x4 fx_matWorldViewProj : WORLDVIEWPROJ;

struct VS_INPUT
{
      float4 f4Position : POSITION0;
      float4 f4Color : COLOR0;
};

struct VS_OUTPUT
{
      float4 f4Position : POSITION0;
      float4 f4Color : COLOR0;
};

struct PS_OUTPUT
{
      float4 f4Color : COLOR0;
};

// muokkaa sisääntulevaa verteksitietoa
void vertex_shader(in VS_INPUT IN, out VS_OUTPUT OUT)
{
      OUT.f4Position = mul(IN.f4Position, fx_matWorldViewProj);
      OUT.f4Color = IN.f4Color;
}

// muokkaa verteksivarjostimen palauttama tieto ja lähetä se laiteelle
//pikseli kerrallaan.
void pixel_shader(in VS_OUTPUT IN, out PS_OUTPUT OUT)
{
      float4 fColor = IN.f4Color;
      OUT.f4Color = clamp(fColor, 0, 1);
}

//määritellään tekniikka verteksi- ja pikselivarjostimen jokaiselle
//passille eli pyyhkäisylle
technique simple
{
      pass p0
      {
            //määrittele verteksi- ja pikselivarjostin
            vertexshader = compile vs_1_1 vertex_shader();
            pixelshader = compile ps_2_0 pixel_shader();
      }
}
```

Kuvio 5.30. Esimerkki varjostin (Cawood & McGee 2007, 45-46.)

Verteksivarjostin (Vertex Shader) suorittaa toimintoja jokaiselle verteksille, jotka se saa parametrina. Verteksivarjostimella voidaan muokata ja piirtää kaikkia verteksin ominaisuuksia. Sitä voidaan käyttää verteksikohtaisen valaistuksen toteuttamiseen, paikkatietojen ja väritietojen muokkaamiseen sekä kuvakoordinaattien säätämiseen. Verteksivarjostin voi muokata ky-

seisiä tietoja erilaisten syötteiden kuten muunnosten ja filtteröinnin avulla. Muutosten jälkeen väri- ja tekstuuripaluutieto voidaan palauttaa pikselivarjostimelle. Paikka- ja normaalitiedot voidaan myös lähettää pikselivarjostimelle, jos näitä tietoja täytyy muokata. Näitä tietoja ei voida muokata enää sen jälkeen, kun tieto lähtee verteksivarjostimesta. Verteksivarjostimen pitää vähintään palauttaa paluutietona paikkatiedot. Pikselivarjostimelle tulevat elementit interpoloidaan ennen kuin ne lähetetään pikselivarjostimelle. (Cawood & McGee 2007, 46)

Pikselivarjostimet (Pixel Shader) muuntavat verteksivarjostimelta saatua verteksitietoa värilliseksi pikselitiedoksi. Pikselivarjostin ei voi muokata vektorin paikka- tai normaalitietoja, mutta se voi suorittaa pikselikohtaisia toimintoja ja muutoksia liittyen pikselin valaistukseen, väriin, tekstuuriin ja sekoitukseen. Paljon tietoa käsiteltäessä pikselikohtaisten muutosten tekeminen on raskaampaa kuin verteksikohtaisten. Efekti, kuten valaistus näyttää paremmalta, jos se tehdään pikselivarjostimella, vaikka se on raskaampi tapa. Joskus on mahdollista, että saatu lopputulos on tehonmenetyksen arvoinen. Pikselivarjostin tulostaa pikselit yksitellen näytönohjaimelle, kun kaikki muutokset on tehty. (Cawood & McGee 2007, 46)

Tekniikka (technique) määrää käytettävän verteksi- ja pikselivarjostimen jokaisen pyyhkäisyn yhteydessä pikseleitä piirtäessä. Useimmissa tapauksissa piirto suoritetaan yhdellä pyyhkäisyllä. Esimerkiksi multitekstu rointia käytettäessä joudutaan yleensä käyttämään useampaa pyyhkäisyä. (Cawood & McGee 2007, 46)

5.4.3 High Level Shader Language (HLSL)

Suurin osa XNA peleissä käytetyistä varjostimista on kirjoitettu Microsoftin High Level Shader Language (HLSL) -kielellä. HLSL muistuttaa melko paljon C -ohjelmointikielen syntaksia. XNA ohjelmoijan on helppo lähestyä sitä, koska C# -ohjelmointikieli kuuluu C-ohjelmointikieliperheeseen. Varjostinkoodit voidaan myös kirjoittaa konekielellä, mutta sen syntaksi on vaikeaa ja se ei ole yhteensopiva kaikkien näytönohjaimien kanssa.

Aikoinaan peliohjelmoijat kirjoittivat varjostinkoodit ainoastaan konekielellä, mutta huono yhteensopivuus eri näytönohjainten välillä aiheutti paljon ongelmia. Näytönohjainvalmistajilla kuten NVIDIA:lla, ATI:lla ja muilla on vastaavanlaiset konekielikirjastot, mutta erilaisuudet eri näytönohjainvalmistajien näytönohjainten välillä aiheuttavat joskus varjostimien yhteensopivuusongelmia. Tästä johtuen pelit, jotka käyttävät tietyn näytönohjainvalmistajan

konekielikirjastoa eivät välttämättä toimi yhtä hyvin sellaisissa tietokoneissa, joissa on toisen valmistajan näytönohjain. Kehittäessä peliä XBOX 360:lle riittää, kun käyttää viimeisintä versiota HLSL -ohjelmointikielestä, mutta kehitettäessä peliä PC:lle tulisi perehtyä paremmin eri näytönohjainvalmistajien eroavaisuuksiin näytönohjainten välillä. Microsoft vaatii, että XNA alustalla toimiva näytönohjain tukee varjostin mallia 2.0, mutta myös varjostinmallilla 1.1 kirjoitetut varjostimet toimivat XBOX 360:ssa ja PC:ssä. (Cawood & McGee 2007, 47)

Verteksi- ja pikselivarjostimien paluutiedot ja syötteet voidaan lähettää parametrilistalla tai paremetritietueilla. Varjostinmerkkioppia käytetään, kun sidotaan varjostinsyötetietoja verteksidataan, mikä saadaan XNA koodista. Varjostinmerkkioppia käytetään myös, sidottaessa syöte- ja paluutietoja eri varjostimien välillä. Varjostinmerkkiopin avulla käyttäjä voi rakentaa haluamiaan tietorakennejoukkoja ja -kokoelmia, joita käytetään liikuteltaessa erilaisia syöte- ja paluutietoja XNA koodin, varjostimien ja Graphics Pipelinen välillä. Käyttäjä voi määrittää oman varjostinmerkkiopin värille, tekstuurikoordinaateille, normaalivektoreille, paikkatiedoille ja monille muille tiedoille. (Cawood & McGee 2007, 47)

Yleisimmin käytettyjä verteksivarjostimen syötetietojen merkkioppeja, joilla voidaan syöttää verteksiominaisuuksia verteksivarjostimelle ovat:

```
COLOR[n]        // väri
NORMAL[n]       // normaaliverktori
POSITION[n]     // verteksipaikkatieto
PSIZE[n]        // pisteen koko pistespriteille
TEXCOORD[n]     // tekstuurikoordinaatit
```

[n]:n osoittama numero määrittää kyseisen tietotyypin indeksin. (Cawood & McGee 2007, 48)

Verteksivarjostimen paluutietojen merkkioppi osoittaa tietoon, mikä lähetetään verteksivarjostimelta pikselivarjostimelle tai Graphics Pipelineen. Pikselivarjostimella ja Graphics Pipelinella voidaan suorittaa erilaisia toimintoja verteksivarjostimen paluutiedoille. Merkkiopilla sidotaan tiedot, jotka siirtyvät verteksivarjostimen ja pikselivarjostimen välillä. Muutamia yleisimmin käytettyjä verteksivarjostimien paluutietojen merkkioppeja ovat: (Cawood & McGee 2007, 48)

```
COLOR[n]        // väri
POSITION[n]     // verteksipaikkatieto
PSIZE[n]        // pisteen koko pistespriteille
TEXCOORD[n]     // tekstuurikoordinaatit
```

Pikselivarjostin voi muokata väri- ja tekstuuritietoja, jotka se saa verteksivarjostimelta. Pikselivarjostimelle voidaan lähettää myös paikkatietoja, mikäli halutaan suorittaa laskennallisia toimintoja erilaisten efektien saavuttamiseksi, kuten esimerkiksi valaistusefektit. Pikselivarjostimella ei voi muokata verteksivarjostimelta saatuja paikkatietoja. Seuraavat tiedot ovat muokattavissa pikselivarjostimella: (Cawood & McGee 2007, 48)

```
COLOR[n]        // väri
TEXCOORD[n]     // tekstuurikoordinaatit
```

Yleisimmissä tilanteissa pikselivarjostimelta palautuu ainoastaan pikselin väriarvo, siksi yleisin pikselivarjostimen paluutiedon merkkioppi on: (Cawood & McGee 2007, 48)

```
COLOR[n]        // väri
```

Tarkasteltaessa HLSL koodia voidaan huomata, että varjostimien tietotyypit ovat melko samanlaisia syntaksiltaan XNA tietotyyppien kanssa. Taulukossa 5.2 on vertailuja XNA tietotyyppien ja HLSL tietotyyppien välillä. (Cawood & McGee 2007, 49)

XNA tietotyyppi	HLSL tietotyyppi
Matrix	float4x4
Texture2D	Texture
struct Mystruct {public Mystruct(int I) {}};	Struct
int	int
Float	float
Vector2	float2 // Taulukko kahdella sarakkeella
Vector3	float3 // Taulukko kolmella sarakkeella
Vector4	float4 // Taulukko neljällä sarakkeella
Color	float3 (Väritiedot ilman alpha-arvoa) float4(Väritiedot alpha-arvon kanssa)

Taulukko 5.2, XNA ja HLSL -tietotyyppien vertailuja. (Cawood & McGee 2007, 49)

HLSL Intrinsic funktiot

HLSL tarjoaa monia funktioita. Nämä funktiot on dokumentoitu Microsoftin MSDN:n Internet-sivulla, osoitteessa:

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c_dec_2004/directx/graphics/reference/HighLevelLanguageShaders/IntrinsicFunctions/IntrinsicFunctions.asp

(Cawood & McGee 2007, 49)

Taulukossa 5.3, on selitettynä muutamien HLSL funktioiden toiminta.

HLSL Intrinsic funktio	Syötteet	Komponenttityyppi	Paluutiedot
abs(a)	a on skalaari, vektori tai matriisi.	float, int	Absoluuttinen arvo a:sta.
clamp(a, min, max)	clamp(a, min, max)	float, int	Puristettu arvo a:sta.
cos(a)	a on skalaari, vektori tai matriisi.	float	Kosini arvo a:sta.
dot(a , b)	a ja b ovat vektoreita.	float	skalaari vektori (dot product).
mul(a , b)	a ja b voivat olla vektoreita tai matriiseja.	float	Matriisi tai vektori, riippuen syötetiedoista.
normalize(a)	a on vektori.	float	yksikkövektori.
pow(a , b)	a on skalaari, vektori tai matriisi. b on potenssi.	a on float, b on integer	a^b
Saturate(a)	a on skalaari, vektori tai matriisi.	a on float	a puristettuna 0 ja 1 väliin.
sin(a)	a on skalaari, vektori tai matriisi.	float	Sini arvo a:sta.
tan(a)	a on skalaari, vektori tai matriisi.	float	Tangentti arvo a:sta.
tex2D(a , b)	a on Sampler2D. b on vektori.	a on Sampler2D, b on 2D float	Vektori

Taulukko 5.3, HLSL Intrinsic funktiot. (Cawood & McGee 2007, 50)

Flow Control syntaksi

Varjostimien syntaksi ja ehtorakenteet ovat C -ohjelmointikielen kaltaisia. Silmukkarakenteista tuettuja ovat for, do-while ja while -silmukat. HLSL if-else ehtorakenne on samanlainen kuin C -ohjelmointikielessä. (Cawood & McGee 2007, 49)

5.4.4 Varjostinviitteen lisääminen XNA Projektiin

Käyttääkseen varjostimia XNA projekteissa, käyttäjän pitää ladata ne ja luoda niistä viittaukset. XNA kehitysalusta tekee tästä toimenpiteestä helpon tarjoamalla Effect -luokan, jonka metodeilla voidaan ladata ja kääntää varjostimia. XNA koodissa voidaan muokata varjostimen muuttujia EffectParameter -luokan avulla. (Cawood & McGee 2007, 49)

Uuden varjostimen lisäys XNA projektiin, tapahtuu painamalla oikeata hiiren painiketta Content tiedoston kohdalla Solution Explorer ikkunassa. Valitaan Add new item, tämän jälkeen aukeaa uusi Add new item -ikkuna. Add new item -ikkunassa valitaan Effect File ja kirjoitetaan haluttu nimi uudelle varjostimelle ja painetaan Add -nappia. Tiedostonimen pitää sisältää (.fx) pääte, jotta sitä voidaan käyttää. Ohjelma luo automaattisesti uuden varjostinluokan sisälle perusluokkamääritykset. Jos käyttäjällä on valmis varjostinkoodi perusluokkamääritykset voidaan poistaa ja lisätä niiden tilalle käyttäjän omaa koodia. (Cawood & McGee 2007, 50)

Olemassa olevia varjostimia voidaan lisätä painamalla Solution Explorer -ikkunassa Content -kansion kohdalla oikeata hiiren painiketta ja valitsemalla listalta Add Existing Item. Avausikkunan auetessa etsitään oma varjostintiedosto (.fx pääte) tiedostohakemistosta, valitaan se ja painetaan Add -nappia. (Cawood & McGee 2007, 50)

Effect -objektin avulla voidaan ladata ja kääntää varjostinkoodia. Lisäksi sillä voidaan hyväksyä eri muuttujamuutokset varjostimella ja lähettää vektoritietoa varjostimelle. Effect -objektin määrittely, alustus on esitetty kuviossa 5.31.

Varjostin voidaan ladata Load -metodilla, kun varjostimen viittaus on tehty XNA projektiin. HLSL varjostintiedostot nimetään automaattisesti .fx päätteellä, mutta varjostinta ladatessa käytetään pelkästään varjostimen tiedostonimeä ilman päätettä. Varjostinkoodin lataus Effect -objektiin on esitetty kuviossa 5.31.

EffectParameter -objektit mahdollistavat varjostimen globaalimuuttujien asettamisen XNA koodista. EffectParameter -objektin luonnin jälkeen voidaan asettaa se viittaamaan Effect -luokan Parameters -kokoelmassa olevaan globaalimuuttujaan. Effect -luokan Parametres -kokoelma pitää sisällään kaikki kyseisen Effect -objektin globaalimuuttujat. Effect -luokan Parametres -kokoelma on indeksoitu objektien nimen mukaisesti. EffectParameter -objekti voidaan asettaa viittaamaan tiettyyn globaalimuuttujaan. Tämä on esitetty kuviossa 7.3.

EffectParameter -objektin määrittelyn ja alustuksen jälkeen voidaan muokata varjostimen muuttujan arvoa kaksivaiheisella toimenpiteellä. Aluksi asetetaan uusi arvo käyttäen SetValue -metodia. Uusiarvo muuttujan tulee olla samaa tietotyyppiä, kuin varjostimen muuttuja johon EffectParameter -objekti viittaa. Uuden arvon sijoittamisen jälkeen pitää hyväksyä varjostimen muutokset ajamalla CommitChanges -metodi (Kuvio 5.31).

```
//Määritellää Effect ja EffectParameter luokat Game luokan alkuun

private EffectParameter effectParameter;
private Effect effect;


//Ladataaan varjostinkoodi Effect luokkaan Initialize-metodissa
effect =
content.Load<Effect>(@"Tiedostopolku\varjostimen_nimi");

//Asetetaan EffectParameter viittaamaan haluttuun muuttujaan ladatun
//varjostinkoodin sisältä Initialize-metodissa
effectParameter = effect.Parameters["globaalimuuttujan_nimi"];

//Päivitetään muuttujan arvo johon EffectParameter objekti viittaa
effectParameter.SetValue(Uusiarvo);

//Hyväksytään muutokset varjostimella
effect.CommitChanges();
```

Kuvio 5.31. Effect ja -Parameter luokan käyttö.(Cawood & McGee 2007, 50-51)

Yksinkertainen varjostinesimerkki

Liitteessä 1 on esitetty esimerkkipeliprojekti ja kaikki siihen liittyvät peliprojektin tiedostot. Esimerkki peliprojekti sisältää yksinkertaisen varjostimen IntroShader.fx , joka palauttaa primitiivipinnan. Primitiivipinta käyttää verteksejä, jotka pitävät sisällään väri- ja paikkatietoja. Esimerkki peliprojektissa muutetaan neliöpinnan sinistä väriarvoa ajastimella 0 ja 1 arvojen välillä, jolloin saadaan aikaan pinnan vilkkuminen. Lisäksi neliöpinnan paikkatietoja X -akselilla muutetaan ajastimella, jotta saadaan aikaan edestakaista liikettä kyseiseen pintaan. (Cawood & McGee 2007, 52)

6 PELIMAAILMA

Kaikki pelimaailman objektit tulisi teksturoida. Tämä pitää sisällään kaiken kasvillisuudesta ihmisiin. Jos esineitä ja ihmisiä ei teksturoida hyvin, peli ei näytä oikealta. (Cawood & McGee 2007, 92)

Tekstuurit ovat kuvia, jotka asetetaan erilaisten objektien pinnoille. XNA tarjoaa kehittäjille laajan valikoiman erilaisia määritteitä teksturoinnille, joilla saadaan aikaiseksi hienoja efektejä. Esimerkiksi tekstuureita voidaan värittää, hajottaa, sekoittaa ja muuntaa suoraan ajon aikana. Ottaen huomioon teksturoinnin tärkeyden, siihen on panostettu paljon XNA:ssa. XNA tukee .bmp, .dds, .dip, .hdr, .jpg, .pfm, .png, .ppm ja .tga kuvatiedostoformaatteja tekstuureille. (Cawood & McGee 2007, 92)

UV koordinaatit määrittelevät pisteen tekstuurissa, joista käytetään usein nimitystä tekstuurikoordinaatti. Tekstuurikoordinaatit eroavat X, Y ja Z koordinaateista, koska tekstuuri on kaksiulotteinen objekti, joka on sijoitettu kolmiulotteiseen polygoniin. Tekstuurin kaksiulotteinen koordinaattidata on varastoitu verteksiin jokaisen X, Y ja Z koordinaatin kanssa. Kun tekstuuri kiinnitettään suorakulmaiseen objektiin, kumpikin U ja V koordinaatit saavat arvon nollan ja yhden väliltä. (Cawood & McGee 2007, 92)

Tekstuureita ladataan ja muokataan käyttämällä Texture2D -luokkaa. Tämä luokka luodaan käyttäen kuvion 6.1 syntaksia. (Cawood & McGee 2007, 92)

XNA:n ContentManager luokkaa käytetään lataamaan peliin sisältöä kuten kuvia. ContentManagerin Load -metodilla saadaan ladattua kuva Texture2D -objektiin, kun kuvat ovat liitetty projektiin. Load -metodi tarvitsee vain kuvan nimen ja tiedostopolun missä se sijaitsee. Kuvan tiedostopäätettä ei välttämättä tarvita. Kuviossa 6.1 ladataan kuva peliprojektin Kuvat -kansiosta. (Cawood & McGee 2007, 93)

```
Texture2D textureObject;
textureObject = content.load<Texture2D>(".\\Kuvat\\kuvantiedosto");
```

Kuvio 6.1. Tekstuurin alustus ja lataaminen Texture2D luokkaan.

Teksturointi toteutetaan varjostimissa. Teksturointia varten tarvittava koodi on samantapaista kuin aikaisemmassa Varjostimet kappaleessa esitetty koodi. Kuitenkin joitakin muutoksia tarvitaan teksturoinnin mahdollistamiseksi. Muutokset ovat:

- Globaali tekstuuri muuttuja
- Näyteobjekti tekstuurin suodatusta varten
- Verteksivarjostimien syöttö- ja tulostusdata, jotka pitävät sisällään UV -koordinaatit
- Pikselivarjostimen koodi, joka kytkee tekstuuridatan yksittäisiin pikseleihin

6.1 Läpinäkyvät tekstuurit

Jossakin vaiheessa tekstuureiden läpinäkyvyydelle voi tulla tarvetta. Esimerkiksi jos tahdotaan tehdä kuvan taustapikseleistä näkymättömiä, kun taas muu kuva piirretään tavallisesti. Tätä voidaan käyttää esimerkiksi puissa ja ns. HUD -grafiikan piirtämiseen. Tämä efekti voidaan toteuttaa käyttämällä maskia, jota .dds tiedostomuoto käyttää. On myös mahdollista tehdä .png tai .tga kuvatiedosto, joka sisältää läpinäkyviä pikseleitä. Läpinäkyvät pikselit voidaan toteuttaa millä tahansa kuvankäsittelyohjelmalla ja sitten käytetään XNA:n piirtologiikassa sellaista koodia, jolla läpinäkyvät pikselit eivät näy. (Cawood & McGee 2007, 98)

Alphakanavaa voidaan käyttää ”maskin” luomiseen kuvalle, tietyllä värillä. Alpha kanavan tieto säilytetään pikselin viimeisessä bitissä, heti punaisen, vihreän ja sinisen bitin jälkeen. Kun ”alpha blending” ominaisuus on päällä XNA koodissa ja alpha kanava on aktiivinen, ovat kaikki ne pikselit läpinäkyviä, joiden alpha arvo on 0.

Teksturointi esimerkki

Kun piirretään objekti, jolla on tekstuureja, GraphicsDevice -luokan täytyy hakea tarvittava data oikeaoppisessa muodossa. Uusi VertexDeclaration -luokka alustetaan, jotta grafiikkalaite voi myöhemmin hakea oikean sijainnin, värin ja UV -koordinaatit. Myöhemmin, Initialize -metodissa, tarvitaan koodi, joka alustaa VertexDeclaration -luokan VertexPositionColorTexture formaattiin. Lisäksi lisätään muuttujat, joilla Texture2D -luokat tunnistetaan niitä piirrettäessä. Jotta jokainen tekstuurin kuva voitaisiin varastoida, tarvitaan Texture2D -luokat niitä

varten. Esimerkissä käytetään kolmea kuvaa, joilla teksturoidaan maa ja kaksi seinää. Ne ladataan projektiin Load -metodissa pelin käynnistyessä. Verteksien tiedot säilytetään verteksimuuttujassa mVert. Muuttuja tulee alustaa niin, että se pystyy säilömään datan kaikista vertekseistä. Kuvio 6.2 näyttää muuttujien alustuksen. (Cawood & McGee 2007, 99-100)

```
private VertexDeclaration mVertPosTexColor1;
mVertPosTexColor1 = new VertexDeclaration(gfx.GraphicsDevice,
VertexPositionColor TexturePositionColorTexture.VertexElements);

const int BACKWALL = 0;
const int GROUND = 1;
const int SIDEWALL = 2;

private Texture2D mTexGround;
private Texture2D mTexBackwall;
private Texture2D mTexSidewall;
mTexGround = Content.Load<Texture2D>(".\\Images\\ground");
mTexBackwall = Content.Load<Texture2D>(".\\Images\\backwall");
mTexSidewall = Content.Load<Texture2D>(".\\Images\\sidewall");
private VertexPositionColorTexture[] mVert = new
VertexPositionColorTexture[4];
```

Kuvio 6.2. Tekstuuri muuttujien alustukset. (Cawood & McGee 2007, 99-100)

Metodi init_surface alustaa verteksit sijainnilla, värillä ja UV koordinaateilla, joita käytetään neliön muotoisen alueen tekoon, jolla peitetään jokainen seinä. UV koordinaatit alustetaan niin, että U arvo menee X akselin mukaisesti ja V arvo Z akselin mukaisesti. (Cawood & McGee 2007, 101)

```
private void init_surface()
{
      Vector2 uv = new Vector2(0.0f, 0.0f);
      Vector3 pos = new Vector3(0.0f, 0.0f, 0.0f);
      Color color = Color.White;

      // Asetetaan verteksien pinnat uv:lla, positiolla ja värillä
      uv.X = 1.0f; uv.Y = 1.0f; pos.X = -BOUNDARY; pos.Y = 0.0f; pos.Z
      = -BOUNDARY;

      mVert[0] = new VertexPositionColorTexture(pos, color, uv);//front
      right

      uv.X = 1.0f; uv.Y = 0.0f; pos.X = -BOUNDARY; pos.Y = 0.0f; pos.Z
      = BOUNDARY;

      mVert[1] = new VertexPositionColorTexture(pos, color, uv);//back
      right

      uv.X = 0.0f; uv.Y = 1.0f; pos.X = BOUNDARY; pos.Y = 0.0f; pos.Z =
      -BOUNDARY;

      mVert[2] = new VertexPositionColorTexture(pos, color, uv);//front
      left
```

```
        uv.X = 0.0f; uv.Y = 0.0f; pos.X = BOUNDARY; pos.Y = 0.0f; pos.Z =
        BOUNDARY;

        mVert[3] = new VertexPositionColorTexture(pos, color, uv);//back
        left
}
```

Kuvio 6.3. init_surface metodi. (Cawood & McGee 2007, 100 - 101)

Tämän lisäksi tarvitaan draw_surface -metodi. Jokaisella kerralla, kun metodia kutsutaan, lisätään kutsuun parametri, jolla tunnistetaan mitä pintaa ollaan piirtämässä. Metodi käyttää perinteistä ”switch case” rakennetta valitsemaan tietyt muutokset ja tekstuurit. Kun tekstuuri on valittu, se asetetaan varjostimeen käyttämällä Effectparameter -luokan SetData -metodia. Kun kumulatiiviset muunnokset on asetettu, WordViewProjection -matriisi asetetaan TextureShader:iin käyttämällä uutta EffectParameter -luokkaa mfxTex_WVP. WorldViewProjection -matriisia käytettään jokaisessa pinnoitteessa, jotta ne voidaan nähdä oikealla tavalla kamerasta käsin. Kun varjostinmuuttujat on asetettu kutsutaan CommitChanges -metodia, jotta muutokset menevät perille. Käyttämällä <VertexPositionColorTexture> viittausta DrawUserPrimitivies -metodin yhteydessä varmistetaan, että tekstuuri lisätään objektiin. Kuvio 6.4 näyttää draw_surface -metodin listauksen. (Cawood & McGee 2007, 101)

```
private void draw_surface(int iSurface)
{
        // 1: Määrittele matriisit
        Matrix matIdentity, matScale, matTransl, matYrot, matXrot;

        // 2: Alusta matriisit
        matIdentity = Matrix.Identity;
        matTransl = Matrix.CreateTranslation(0.0f, -0.9f, 10.0f);
        matScale = Matrix.CreateScale(0.1f, 0.1f, 0.1f);
        matXrot = Matrix.CreateRotationX(-(float)Math.PI / 2.0f);
        matYrot = Matrix.CreateRotationY(0.0f);

        // Tee muunnokset ja aseta tekstuurit jokaiselle objektille
        switch (iSurface)
        {
                case GROUND:    // Maa keskitettynä
                matXrot = Matrix.CreateRotationX(0.0f);
                mfxTexture.SetValue(mTexGround);   // Aseta tekstuuri
                break;

                case BACKWALL:  // Pyöritä -90 astetta X akselilla ja
                sierra taakse sekä ylöspäin
                matTransl = Matrix.CreateTranslation(0.0f, 0.70f, 11.6f);
                mfxTexture.SetValue(mTexBackwall); // Aseta tekstuuri
                break;

                case SIDEWALL:  // Pyöritä -90 astetta X akselilla sekä +90
                astetta Y akselilla. Siirrä vasemmalle ja ylöspäin
```

```
            matYrot = Matrix.CreateRotationY((float)Math.PI / 2.0f);
            matTransl = Matrix.CreateTranslation(1.6f, 0.70f, 10.0f);
            mfxTexture.SetValue(mTexSidewall); // Aseta tekstuuri
            break;
    }


     // 3: Rakenna kumulatiivinen maailma matriisi
     // Skaalaa, pyöritä, kierrä ja muunna
     mMatWorld = matIdentity * matScale * matXrot * matYrot *
     matTransl;

     // 4: Syötä wvp matriisi shaderiin
     mfxTex_WVP.SetValue(mMatWorld * mMatView * mMatProj);
     mfxTex.CommitChanges(); // Hyväksy muutokset

     // 5: Piirrä objekti
     gfx.GraphicsDevice.VertexDeclaration = mVertPosTexColor1;

     gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColorTexture>
     (PrimitiveType.TriangleStrip, mVert, 0, 2);     // Käytä
     tekstuurityyppiä

}
```

Kuvio 6.4. draw_surface -metodi. (Cawood & McGee 2007, 101)

Näin ollen nyt voidaan piirtologiikassa kutsua piirtometodia jokaiselle objektille (Kuvio 6.5)

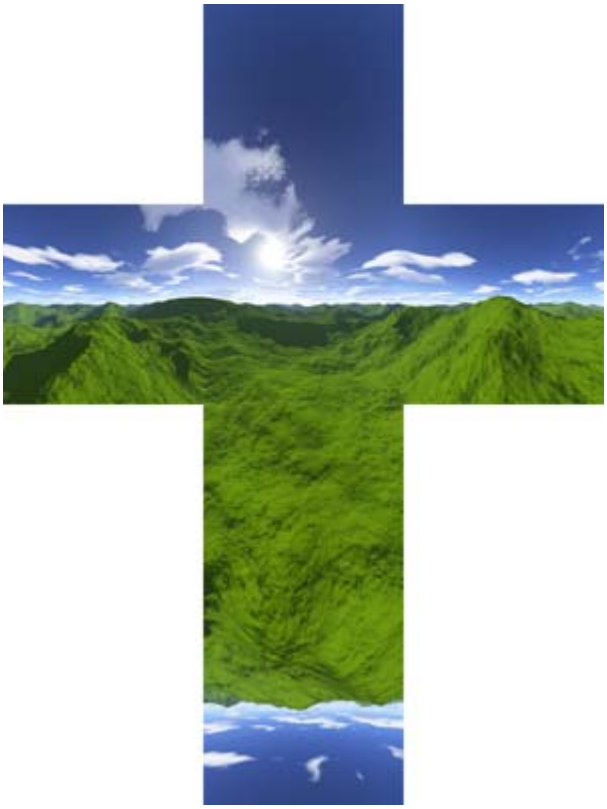
```
draw_surface(GROUND);
draw_surface(BACKWALL);
draw_surface(SIDEWALL);
```

Kuvio 6.5 Objektien piirtologiikan kutsut.

6.2 Taivaan ja horisontin toteutus

Rakennetta, jolla esitetään taivas ja horisontti kutsutaan usein nimellä skybox. Kun skybox on rakennettu oikein, se on saumaton ja näin ollen ei voida helposti huomata mistä se alkaa ja mihin se loppuu. (Cawood & McGee 2007, 110)

Skybox voidaan rakentaa kuudesta erillisestä kuvasta. Kuviossa 6.6 kaikki kuusi kuvaa ovat samassa tiedostossa, josta erilliset kuvat otetaan ja asetetaan eri pinnoille rakentamaan maa, taivas ja horisontti. Skybox tekstuureita löytyy paljon Internetistä, joita voidaan käyttää vapaasti ei-kaupalliseen tarkoitukseen. Myös Terragen maastonteko-ohjelmaa voidaan hyödyntää skybox kuvien tekoon.



Kuvio 6.6. Skybox tekstuuri. (OpenSG, 2005)

Tässä esimerkissä yhdistetään 6 erillistä kuvaa muodostamaan saumaton skybox. Esimerkissä oletetaan, että kuvat ovat projektin juurihakemistossa ja ne on referoitu Visual Studioon Solution Explorerin kautta. (Cawood & McGee 2007, 110)

Ensin täytyy määritellä tekstuuriluokat, joissa kuvat säilötään. LoadGraphicsContent -metodissa ladataan kuvat tekstuuriobjekteihin (Kuvio 6.7).

```
Texture2D mTexFront, mTexBack, mTexGround, mTexLeft, mTexRight, mTexSky

mTexFront = Content.Load<Texture2D>(".\\Images\\front");
mTexBack = Content.Load<Texture2D>(".\\Images\\back");
mTexLeft = Content.Load<Texture2D>(".\\Images\\left");
mTexRight = Content.Load<Texture2D>(".\\Images\\right");
mTexGround = Content.Load<Texture2D>(".\\Images\\ground2")
mTexSky = Content.Load<Texture2D>(".\\Images\\sky");
```

Kuvio 6.7. Tekstuuriluokkien määritykset. (Cawood & McGee 2007, 115)

Seuraava init_skybox -metodi sisältää kaiken koodin tarvittavien verteksien alustukseen, jotta skybox voidaan piirtää. Jokaisella kerralla, kun näitä koordinaatteja käytetään piirtämään paneelia, ne täytyy muuntaa sijaintiin nähden. (Cawood & McGee 2007, 116)

Lisäksi init_skybox -metodin (Kuvio 6.8) kutsu lisätään pelin Initialize -metodiin.

```
private void init_skybox()
{
      Vector3 pos = new Vector3(0.0f, 0.0f, 0.0f);
      Vector2 uv  = new Vector2(0.0f, 0.0f);

      const float max = 0.997f; // Poista valkoinen reunus ylälaidasta
      const float min = 0.003f; // Poista valkoinen reunus alalaidasta

      // Aseta position, kuva ja väri data
      pos.X = +EDGE; pos.Y = -EDGE; uv.X = min; uv.Y = max; //Bottom L
      mVertSkybox[0] = new VertexPositionColorTexture(pos, Color.White,
      uv);

      pos.X = +EDGE; pos.Y = +EDGE; uv.X = min; uv.Y = min; //Top L
      mVertSkybox[1] = new VertexPositionColorTexture(pos, Color.White,
      uv);

      pos.X = -EDGE; pos.Y = -EDGE; uv.X = max; uv.Y = max; //Bottom R
      mVertSkybox[2] = new VertexPositionColorTexture(pos, Color.White,
      uv);

      pos.X = -EDGE; pos.Y = +EDGE; uv.X = max; uv.Y = min; //Top R
      mVertSkybox[3] = new VertexPositionColorTexture(pos, Color.White,
      uv);
}
```

Kuvio 6.8. init_skybox metodi. (Cawood & McGee 2007, 116)

Jotta jokainen skyboxin paneeli piirrettäisiin, täytyy sille vielä rakentaa logiikka. Tämä metodi (Kuvio 6.9) iteroi kaikkien skyboxin paneeleiden läpi, asettaa ne paikoilleen ja piirtää ne oikealla tekstuurilla. (Cawood & McGee 2007, 117)

```
private void draw_skybox()
{
      const float kfDrop = -1.2f;
      // 1: Määrittele matriisit ja aseta vakiot
      Matrix matIdentity  = Matrix.Identity
      Matrix matRotY = Matrix.CreateRotationY(0.0f);
      Matrix matRotX = Matrix.CreateRotationX(0.0f);
      Matrix matTransl = Matrix.CreateTranslation(0.0f, 0.0f, 0.0f);
      Matrix matScale = Matrix.CreateScale(1.0f, 1.0f, 1.0f);
      Matrix matCam = Matrix.CreateTranslation(cam.m_vPos.X,0.0f,
      cam.m_vPos.Z);

      // 2: Aseta muunnokset ja teksturoi jokainen paneeli/seinä

      for (int i = 0; i < 5; i++) // front, right, left, right, & sky
      {
            switch (i)
            {
            case 0: // Takaseinä
            matTransl = Matrix.CreateTranslation( 0.0f, kfDrop, EDGE);
            mfxTexture.SetValue(mTexBack);
```

```
            break;

            case 1: // Oikea seinä
            matTransl = Matrix.CreateTranslation(-EDGE, kfDrop, 0.0f);
            matRotY = Matrix.CreateRotationY(-(float)Math.PI / 2.0f);
            mfxTexture.SetValue(mTexRight);
            break;

            case 2: // Etu seinä
            matTransl = Matrix.CreateTranslation(0.0f, kfDrop, -EDGE);
            matRotY = Matrix.CreateRotationY((float)Math.PI);
            mfxTexture.SetValue(mTexFront);   break;

            case 3: // Vasen seinä
            matTransl = Matrix.CreateTranslation( EDGE, kfDrop, 0.0f);
            matRotY   = Matrix.CreateRotationY((float)Math.PI / 2.0f);
            mfxTexture.SetValue(mTexLeft);    break;

            case 4: // Taivas
            matTransl= Matrix.CreateTranslation(0.0f,EDGE+kfDrop,0.0f);
            matRotX = Matrix.CreateRotationX(-(float)Math.PI / 2.0f);
            matRotY = Matrix.CreateRotationY( 3.0f*(float)Math.PI /
            2.0f);
            matScale  = Matrix.CreateScale(1.0f, 1.0f, 1.0f);
            mfxTexture.SetValue(mTexSky);
            break;
            }

            // 3: Rakenna maailmamatriisi
            mMatWorld =
            matIdentity*matScale*matRotX*matRotY*matTransl*matCam;

            // 4: Liitä maailmamatriisi shaderiin
            mfxTex_WVP.SetValue(mMatWorld * mMatView * mMatProj);
            mfxTex.CommitChanges();

            // 5: Piirrä objektit
            gfx.GraphicsDevice.VertexDeclaration = mVertPosColorTex;
            gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColorTe
            xture>(PrimitiveType.TriangleStrip, mVertSkybox, 0, 2);
      }
}
```

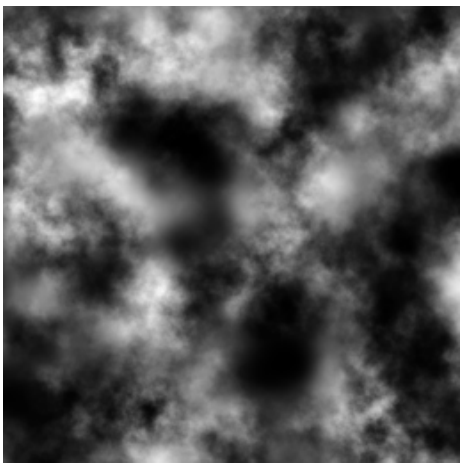
Kuvio 6.9 Skyboxin piirtologiikka. (Cawood & McGee 2007, 117)

Tämän jälkeen metodin kutsu täytyy enää lisätä pelin Draw -metodiin. On olemassa myös toinen tunnettu menetelmä skyboxin tekoon. Tämä tehdään mallintamalla maailman ympärille puolipallon muotoinen objekti ja tämän jälkeen taivastekstuuri asetetaan siihen. Näin saadaan aikaiseksi ns. SkyDome (Cawood & McGee 2007, 119)

6.3 Maaston toteutus

Tässä kappaleessa käsitellään miten voidaan toteuttaa aaltoilevia kenttiä ja niihin liittyvä korkeuden tunnistus. Tällä tavoin pelikamera myötäilee maaston muotoja, ja samalla logiikalla voidaan toteuttaa korkeuden tunnistus pelin eri objekteille. Tässä kappaleessa maasto toteutetaan korkeuskartta -nimisellä (heightmap) tekniikalla. (Cawood & McGee 2007, 412)

Korkeuskartta on kuva, joka varastoi tiedon maaston korkeudesta jokaisessa yksittäisessä pikselissä. Korkeuskartan käyttö maaston luomiseen on suosittua koska suunnittelijat voivat helposti generoida maastoa tavallisella kuvankäsittelyohjelmistolla ja tämä on taas helppo konvertoida 3D-ympäristöön. Kuviossa 6.10 nähdään tyypillinen korkeuskartta kuva. Seuraavassa esimerkissä näytetään miten luodaan ja toteutetaan korkeuskartta käyttämällä kahdeksan bittistä .raw tiedostomuodon kuvaa, joka on harmaasävyinen. Jokainen .raw kuvan pikseli sisältää tiedon korkeudesta väliltä 0 - 255. Nämä tiedot ladataan pelin käynnistyessä taulukkoon myöhempää käyttöä varten. Jokaisen pikselin korkeustietoon päästään käsiksi käyttämällä pikselin rivin ja sarakkeen numerolla. Kun tekniikalla luodaan maastoa 3D-ympäristöön, maasto jaetaan yhtä moneen riviin ja sarakkeeseen kuin kuvakin. Kun kameraa tai muita objekteja asetellaan maastoon, lasketaan objektin korkeus määrätyn rivin ja sarakkeen mukaan. Tästä solusta voidaan sitten palauttaa oikea korkeustaso objektille. (Cawood & McGee 2007, 412)



Kuvio 6.10 Korkeuskartta. (Wikipedia c.)

Uusien korkeuskarttojen tekoon voidaan käyttää mm. Planetside yrityksen Terragen ohjelmistoa. Samalla sovelluksella voidaan myös toteuttaa maastoon liittyvät skybox tekstuurit.

Terragen tarjoaa monia eri tapoja korkeuskartan muodostamiseen. Korkeuskartta voidaan luoda valmiista muotista tai muokata oma. Terragen ohjelmiston kokeiluversio on ladattavissa osoitteesta http://www.planetside.co.uk/terragen/. (Cawood & McGee 2007, 111)

Terragen on yksinkertainen maastonluonti sovellus, jolla saadaan helposti aikaan näyttäviä maisemia. Kun hyvältä näyttävä maasto on luotu, se voidaan tallentaa Landscape ikkunan Export -napista. Tiedosto on tallennettava ensin 8 bittisenä raw -tiedostona, josta se konvertoidaan myöhemmin XNA:n tukemaan muotoon kuten bmp. Korkeuskartta itsessään ei pidä sisällään tekstuuria. Voit käyttää Terragen sovellusta tekstuurin luonnissa tai halutessasi voit käyttää omaa kuvaa maaston teksturointiin.

Esimerkki näyttää miten piirretään yksinkertainen maasto. Ladattavina tiedostoina käytetään vain tekstuuria ja korkeuskarttakuvatiedostoa, jonka perusteella maasto luodaan. Esimerkissä (Kuvio 6.11) käytetään myös kustomoitua Content Pipeline -luokkaa, joka konvertoi korkeuskartan geometriaksi ja luo sen perusteella 3D-mallin. (XNA Creators 2008)

```
[ContentProcessor]

public class TerrainProcessor : ContentProcessor<Texture2DContent,
ModelContent>

{
        const float terrainScale = 3;
        const float terrainBumpiness = 64;
        const float texCoordScale = 0.1f;
        const string terrainTexture = "rocks.bmp";

        /// <summary>
        /// Generoi maaston syötetystä korkeuskartasta
        /// </summary>
            public override ModelContent Process(Texture2DContent
            input, ContentProcessorContext context)
            {
            MeshBuilder builder = MeshBuilder.StartMesh("terrain");

            // Konvertoi syötetyn tekstuurin float muotoon
            input.ConvertBitmapType(typeof(PixelBitmapContent<float>));

            PixelBitmapContent<float> heightfield;
            heightfield = (PixelBitmapContent<float>)input.Mipmaps[0];

            // Luo maaston verteksit
            for (int y = 0; y < heightfield.Height; y++)
            {
                for (int x = 0; x < heightfield.Width; x++)
                {
                    Vector3 position;
```

```
                position.X = (x - heightfield.Width / 2) *
                terrainScale;
                position.Z = (y - heightfield.Height / 2) *
                terrainScale;

                position.Y = (heightfield.GetPixel(x, y) - 1) *
                terrainBumpiness;

                builder.CreatePosition(position);
            }
        }

        // Luo uusi materiaali ja osoita se maaston tekstuuriin.
        BasicMaterialContent material = new BasicMaterialContent();

        string directory =
        Path.GetDirectoryName(input.Identity.SourceFilename);
        string texture = Path.Combine(directory, terrainTexture);

        material.Texture = new
        ExternalReference<TextureContent>(texture);

        builder.SetMaterial(material);

        // Tekstuuri kordinaatit.
        int texCoordId = builder.CreateVertexChannel<Vector2>(

        VertexChannelNames.TextureCoordinate(0));

        // Luo yksittäiset verteksit
        for (int y = 0; y < heightfield.Height - 1; y++)
        {
          for (int x = 0; x < heightfield.Width - 1; x++)
          {
          AddVertex(builder, texCoordId, heightfield.Width,  x, y);

          AddVertex(builder, texCoordId, heightfield.Width, x + 1,
          y);

          AddVertex(builder, texCoordId, heightfield.Width, x + 1,
          y + 1);

          AddVertex(builder, texCoordId, heightfield.Width, x, y);

          AddVertex(builder, texCoordId, heightfield.Width, x + 1,
          y + 1);

          AddVertex(builder, texCoordId, heightfield.Width, x, y +
          1);
          }
        }

        MeshContent terrainMesh = builder.FinishMesh();

        return context.Convert<MeshContent,
        ModelContent>(terrainMesh, "ModelProcessor");
        }


    /// <summary>
```

```
        /// Apumetodi uuden verteksin lisäykseen tekstuuri
        /// kordinaattien kanssa
        /// </summary>
        static void AddVertex(MeshBuilder builder, int texCoordId, int
        w, int x, int y)
        {
            builder.SetVertexChannelData(texCoordId, new Vector2(x, y)
            * texCoordScale);

            builder.AddTriangleVertex(x + y * w);
        }
}
```

Kuvio 6.11. TerrainProsessor esimerkki (XNA Creators 2008)

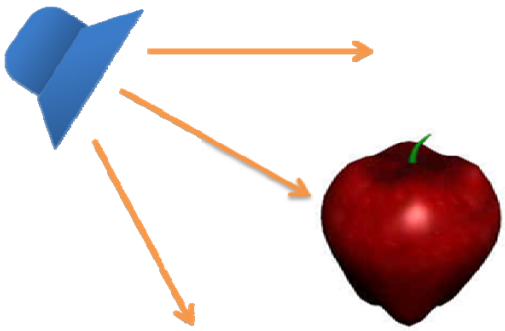
6.4 Valaistus

Pelkästään yksiväristen pintojen ja mallien piirtäminen tuottaa vain tylsän ja lattean 3D -ympäristön. Tekstuureiden lisääminen lisää realismia hieman, mutta maailma saattaa silti näyttää liian steriililtä. Pelaajat eivät välttämättä pysty erottamaan mikä pelimaailmassa on väärin, mutta he pystyvät aistimaan, että jokin ei ole kohdallaan. Valaistus on siis näin ollen yksi tärkeimmistä tekijöistä tehokkaan ja vakuuttavan 3D -ympäristön luontiin. Valaistuksen värejä ja tehoa sekä teksturoimalla pinnat saadaan 3D -objekteihin paljon enemmän syvyyttä ja ne näyttävät realistisimmalta. (Hall 2008, 139)

6.4.1 Valaistustyypit

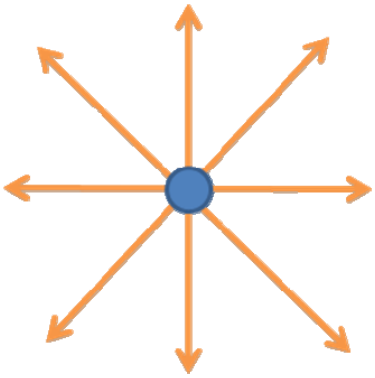
Konseptina valaistus on melko yksinkertainen. Erilaisia valonlähteitä, joilla tosielämän valaistusta voidaan toteuttaa, ovat seuraavat:

Spottivalot sijaitsevat jossain päin pelimaailmaa ja ne valaisevat yhteen tiettyyn suuntaan. Valo kantautuu lähteestä kohdetta päin kartiomaisessa muodossa. Näin valo kattaa pitemmälle mennessä suuremman alueen kuin mitä se tekee lähempänä valon lähdettä (Kuvio 6.12). Esimerkkejä spottivaloista ovat esimerkiksi taskulamppu, kohdevalo, autojen etuvalot ja katuvalot. Pinnat lähempänä valokeilan keskustaa ovat enemmän valaistuneena kuin valokeilan reunoilla olevat pinnat. (Hall 2008, 139)



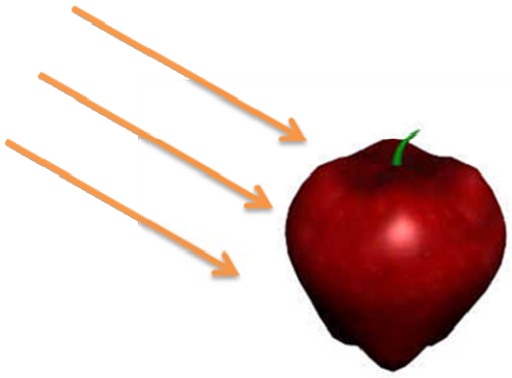
Kuvio 6.12. Spottivalo

Pistevalot loistavat valoaan kaikkiin suuntiin ja valo kattaa suuren alueen (Kuvio 6.13). Esimerkkejä pistevaloista ovat esimerkiksi hehkulamput, kynttilät ja muut avoimet tulet. Pinnat, jotka ovat suoraan valoa vasten, ovat valoisampia kuin pinnat, jotka ovat valosta poispäin. (Hall 2008, 140)



Kuvio 6.13. Pistevalo

Suuntavaloilla ei ole asetettua sijaintia. Tämän valaistuksen valo tulee enemmänkin asetetusta suunnasta, kuin tarkkaan määritetystä sijainnista pelimaailmassa. Suuntavalon valonsäteet osuvat jokaisen mallin pinnoille täysin samassa kulmassa riippumatta missä hahmo tai objekti sijaitsee (Kuvio 6.14). Paras tosimaailman esimerkki suuntavalosta on tietenkin aurinko. (Hall 2008, 140)



Kuvio 6.14. Suuntavalo

On olemassa monia erilaisia valaisumalleja ja jokaisella niillä on etunsa ja haittansa. Pääpiirteittäin realistisemmat valaisutekniikat vaativat enemmän suorituskykyä. Kun yhden pinnan prosessointiin kuluu aikaa enemmän se tarkoittaa, että tietyssä ajassa voidaan käsitellä vähemmän pintoja. (Hall 2008, 140)

6.4.2 Valaisun komponentit

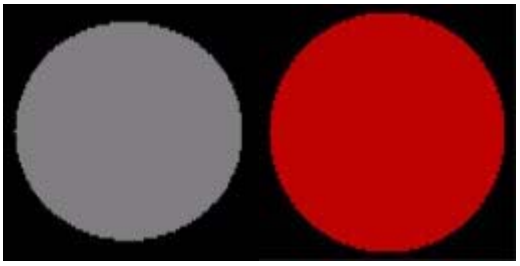
Valaistus, joka näkyy eri objekteissa, voidaan jakaa kolmeen eri peruskomponenttiin. Näitä kolmea tekijää yhdistelemällä eri tavoin voidaan saavuttaa oikeanlaisia realistisia valaisuja virtuaaliseen ympäristöön. On olemassa myös vielä monimutkaisempia ja realistisimpia valaisutapoja kuten heijastus, refraktio, valon ”imeytyminen” eri pintoihin ennen kuin se saavuttaa katsojan silmän jne. Seuraavaksi käsitellään ambient, diffuusio ja speclar valaistuksia, joilla peli saadaan näyttämään riittävän realistiselta. (Hall 2008, 141)

Harvat oikean maailman paikat ovat täysin valottomia. Kirkkaassa päivänvalossa objektit, jotka ovat varjossa, ovat täysin näkyvissä. Himmeästi valaistussa huoneessa voidaan kaikista objekteista nähdä paljon yksityiskohtia, vaikka ne eivät olisikaan lähellä valonlähdettä. Tätä kutsutaan ambient valaistukseksi. (Hall 2008, 141)

Ambient valaistus toteutetaan niin, että 3D mallin pinnan värin annetaan vaikuttaa enemmän viimeiseen lopputulokseen. Kun ambient arvo on nolla, objekti näkyy pelkkänä mustana varjokuvana. Kun ambient arvo maksimoidaan, pinnan väri dominoi viimeistelyä kuvaa ja malli näkyy itseään valaisevana erityisesti silloin, kun sen vieressä on tummempia objekteja. Am-

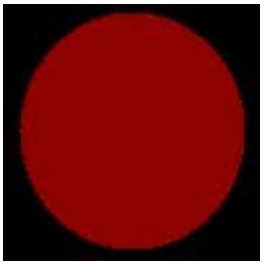
bient valo jakaantuu tasaisesti pinnoille ja siihen eivät vaikuta pinnan normaalivektorit. (Hall 2008, 141)

Kuviossa 6.15 olevat kaksi kuvaa näyttävät materiaalin värin (harmaa), sekä valon värin (kirkkaan punainen). (Mathematics of Lighting, 2008)



Kuvio 6.15. Ambient materiaali ja valo (Mathematics of Lighting, 2008)

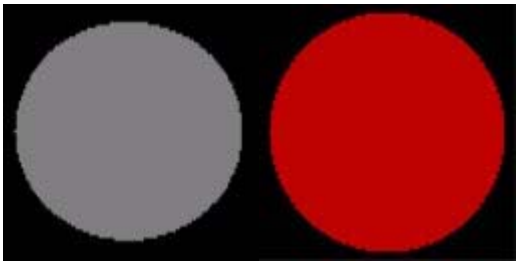
Lopputulos voidaan nähdä alapuolella olevasta kuviosta 6.16. Ambient valo valaisee kaikki objektin verteksit samalla värillä. Se ei ole riippuvainen normaali vertekseistä tai valonsuunnasta. Joten tuloksena nähdään pallo, joka on kuin kaksiulotteinen ympyrä, koska siihen ei luoda minkäänlaista varjostusta. (Mathematics of Lighting, 2008)



Kuvio 6.16. Ambient valon tulos (Mathematics of Lighting, 2008)

Valo kulkee valonlähteestä suoraa linjaa kunnes se osuu johonkin pintaan, josta se heijastuu. Pinnat mitkä ovat valon lähteeseen päin vastaanottavat enemmän valoa ja ne näkyvät kirkkaammin kuin pinnat, jotka ovat poispäin valosta. Tätä kutsutaan diffuusio valaistukseksi. Diffuusiovalon voimakkuuteen vaikuttaa pinnan ja valonlähteen välinen kulma. (Hall 2008, 142 ; Mathematics of Lighting, 2008)

Nämä kaksi kuvaa näyttävät materiaalin värin (harmaa), sekä valon värin (kirkkaan punainen). (Mathematics of Lighting, 2008)



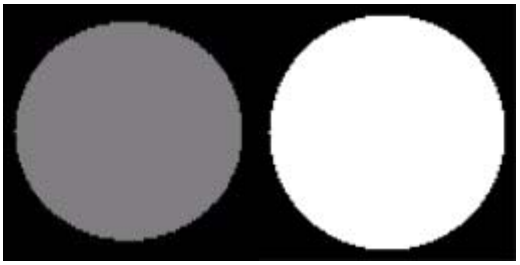
Kuvio 6.17. Diffuusio materiaali ja valo (Mathematics of Lighting, 2008)

Diffuusiovalon tulos nähdään alapuolella olevassa kuviossa 6.18. Kuvan ainoa objekti on pallo. Diffuusio valaistus ottaa huomioon materiaalin ja valon diffuusion värin ja laskee sen perusteella valaistuksen. Tässä valaistuksessa otetaan myös huomioon normaaliverteksit sekä valon suunta, joiden vaikutus lopputulokseen lasketaan käyttäen dot product menetelmää. Tuloksena on näin ollen pallo, jonka toinen puoli on huomattavasti tummempi kuin toinen. (Mathematics of Lighting, 2008)

Kuvio 6.18. Diffuusio valon tulos (Mathematics of Lighting, 2008)

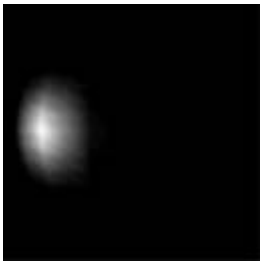
Kun valo osuu kaarevaan pintaan, se hajoaa moneen eri suuntaan. Osa valosta heijastuu suoraan katsojan silmää ja tämä osa loistaa erittäin kirkkaasti. Tätä valaistusta kutsutaan specular valaistukseksi. Efekti on erittäin paikallinen ja se on riippuvainen kulmista valonlähteen, pinnan normaalivektorien ja katsojan silmän eli kameran välillä. Eli efekti voidaan nähdä vain silloin, kun valo tulee valonlähteestä oikeassa kulmassa objektiin ja se heijastuu siitä katsojan silmään. (Hall 2008, 142)

Kuviossa 6.19 näkyvät materiaalin värit mitkä ovat harmaa, sekä valon värin mikä on täysin valkoinen. (Mathematics of Lighting, 2008)



Kuvio 6.19. Specular materiaali ja valo (Mathematics of Lighting, 2008)

Tuloksena saatava Specular valaistuksen alla oleva pallo nähdään kuviossa 6.20.



Kuvio 6.20 Specular valon tulos (Mathematics of Lighting, 2008)

Kun yhdistetään Specular valaistuksen korostus Ambient ja Diffuusio valon kanssa saadaan seuraavanlainen kuvion 6.21 mukainen tulos. Kun kaikki kolme eri valaisutyyppiä ovat käytössä, saadaan tulokseksi kaikista realistisin vaihtoehto. (Mathematics of Lighting, 2008)

Kuvio 6.21 Kolmen valaisutyypin tulos (Mathematics of Lighting, 2008)

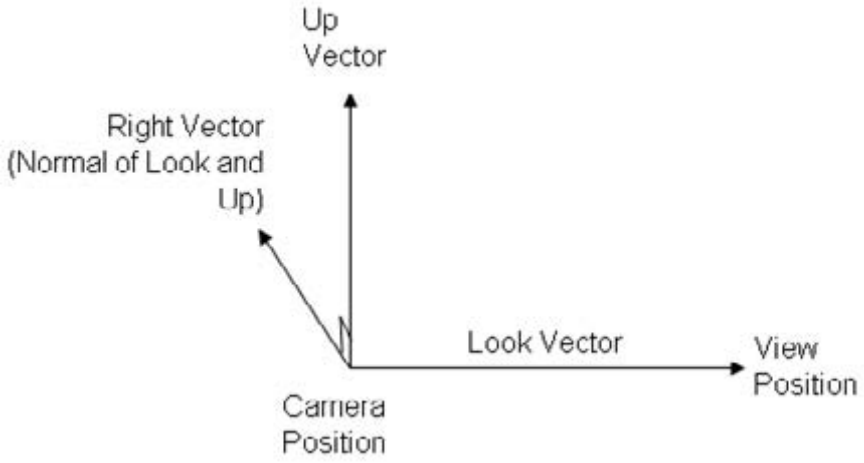
6.5 Kamera

Kamera on hyvin tärkeä osa peliä. Kamera on pelimoottorin sydän, koska sen avulla pelaajalle luodaan kuva virtuaalimaailmaan. Kameran avulla siis määräytyy mitä pelaajan ruudulla

näkyy. Yleensä kamera sisältää logiikan, joka vastaanottaa ja käsittelee pelaajan syötteet ja niiden pohjalta liikuttaa kameraa. Näin pelaaja voi muuttaa näkymäänsä ja liikkua virtuaalimaailmassa. Kamera on siis tärkeä osa sitä, kuinka pelaaja tuntee hallitsevansa peliä tai pelihahmoaan. (Cawood & McGee 2007, 228 ; Hall 2008, 122)

Yleensä kameralogiikka rakennetaan niin, että siihen voidaan vaikuttaa yleisillä kameran vektoreilla ja matriiseilla kuten kuviossa 6.22. Näiden avulla kameraa voidaan hallita ja niiden pohjalta määräytyy myös kameran paikka ja suunta. Jotta kameraa voidaan hallita ja sen pohjalta piirtää kuva virtuaalimaailmasta, pitää tietää kameran sijainti ja suunta sekä kierto. Nämä voidaan määrittää esimerkiksi viidellä vektorilla: (Cawood & McGee 2007, 228)

- View: paikka jossa on kameran fokus
- Position: määrää kameran sijainnin
- Up: määrää kameran pysty asennon
- Look: määrää kameran linssin suunnan
- Right: normaalivektori Look ja Up -vektoreista



Kuvio 6.22 Kameran vektorit

Jotta kamera saadaan toimimaan oikein, käytetään apuna kolmea matriisia. Näiden avulla pelimaailman objektit saadaan piirrettyä oikein eli juuri niin kuin niiden tulee kameralle ja pelaajalle näkyä. Matriisien avulla määräytyvät näkyvät objektit, sekä kulma ja matka, jolla objektit

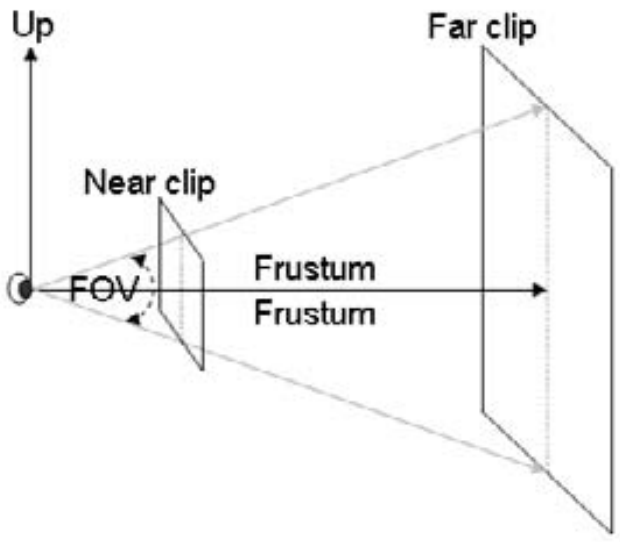
näkyvät. Nämä kolme matriisia ovat maailmamatriisi, näkymämatriisi ja perspektiivimatriisi. (Cawood & McGee 2007, 229)

Maailmamatriisin avulla muunnetaan verteksien ja 3D mallien koordinaatit pelimaailman koordinaateiksi, jotta ne olisivat oikeilla paikoilla 3D pelimaailman avaruudessa. Näkymämatriisi määrää kameran suunnan ja siitä määräytyy mitä pelaaja näkee. Perspektiivimatriisi kuvaa kameran näkymän näkökartion (view frustum) muodossa. Näkökartiolle on määrätty syvyyssuunnassa etu- ja takaraja, jotka ovat lähileikkaus ja takaleikkaus. Näkökartio luodaan käyttäen Matrix -luokan metodia CreatePerspectiveFieldOfView, joka ottaa vastaan viisi parametria. Esimerkki metodin käytöstä ja parametreistä on kuviossa 6.23. Kuviossa 6.24 on esimerkki kameran perspektiivistä, jossa esitety on näkökartio, etu- ja takaleikkaukset. (Cawood & McGee 2007, 229)

```
float fieldOfView; // näkymän kulma
float aspectRatio; // leveys ja korkeus
float nearClip; // ensimmäinen kameralla näkyvä piste syvyyssuunnassa
float farClip; // viimeinen kameralla näkyvä piste syvyyssuunnassa

// Matriisin luonti
Matrix mMatProj = Matrix.CreatePerspectiveFieldOfView(fieldOfView,
                                                      aspectRatio,
                                                      nearClip,
                                                      farClip);
```

Kuvio 6.23. CreatePerspectiveFieldOfView -metodin käyttö



Kuvio 6.24. Kameran perspektiivi

7 SYÖTTEIDEN KÄSITTELY JA ÄÄNET

Syötteiden käsittely on keskeinen osa pelikokemusta. Syöttölaitteita ovat mm. hiiri, näppäimistö ja erilaiset peliohjaimet. XNA:ssa syötteiden käsittely on kohtuullisen helppoa. Microsoft.Xna.Framework.Input -nimiavaruus ja Input -kirjasto helpottavat huomattavasti syötteiden käsittelyä. Kirjasto sisältää tuen hiiren, näppäimistön ja XBOX 360 peliohjaimen syötteiden käsittelylle. Syötteitä ovat esimerkiksi näppäimistön ja hiiren nappien painallukset, hiiren liike, peliohjaimen nappien painallukset sekä ”ohjaintattien” liikkeet. Tämän lisäksi kirjaston avulla voidaan säätää peliohjaimen tärinätoimintoa eli halutussa tilanteessa voidaan täristää pelaajan ohjainta. (Cawood & McGee 2007, 176 ; Nitschke 2007, 169-170)

Syötteiden käsittely pelikehityksessä eroaa normaalista työpöytäohjelmistokehityksestä siten, ettei se ole tapahtumapohjaista. Syötteitä ei siis käsitellä silloin kun käyttäjä antaa syötteen vaan syötteet luetaan jokaisen pelisilmukan yhteydessä. Tämä lähestymistapa voi tuntua tehottomalta, mutta koska pelisilmukka ajetaan läpi useita kymmeniä kertoja sekunnissa, on hyvin epätodennäköistä, että syöte jää käsittelemättä. (Hall 2008, 150-151, 170, 186-187.)

7.1 Hiiri

Monissa PC peleissä käytetään hiirtä pelaajan liikkeiden ohjaamiseen. XNA Frameworkin Input kirjaston avulla on mahdollista lukea hiiren liikkeitä ja painalluksia ja käyttää niitä pelissä. Tiedot hiiren positiosta ja näppäinten tilasta ovat MouseState -objektissa, johon ne haetaan GetState -metodilla jokaisen pelisilmukan yhteydessä. Esimerkki hiiren tilan hausta on kuviossa 7.1. MouseState -objektista voidaan hakea hiiren näppäinten tila, joka voi olla joko painettu tai vapautettu. Lisäksi MouseState -objektista saadaan hiiren sijainti X ja Y akseleille. Koska nykyhiirissä on usein myös rulla, tuki sille on myös lisätty XNA Frameworkiin. Rullan tila saadaan MouseState -objektista ScrollWheelValue -komennolla. (Cawood & McGee 2007, 335 ; Hall 2008, 185-187.)

```
//Haetaan viimeisin hiiren tila
MouseState mouseState = Mouse.GetState();

if (mouseState.LeftButton == ButtonState.Pressed)
{
      //Hiiren vasen näppäin on painettuna
}

//Haetaan hiiren position
int x = mouseState.X;
int y = mouseState.Y;

//Haetaa rulla tila
int scroll = mouseState.ScrollWheelValue;
```

Kuvio 7.1. Hiiren tilan haku

7.2 Näppäimistö

XNA Framework tukee yli sataa normaalin näppäimistön näppäintä. Kaikkia näppäimiä kohdellaan digitaalisina eli näppäimistössä ei ole analogisia näppäimiä. Tämän takia jokaisella näppäimellä on kaksi tilaa: painettu tai vapautettu. Keys -enumista löytyvät kaikki tuetut näppäimet ja ne ovat seuraavat: (Cawood & McGee 2007, 334-335. ; Hall 2008, 169-171.)

A - Z
Add
CapsLock
D0 - D9
Decimal
Delete
Divide
Down
End
Enter
Escape
F1 - F12
Help
Home
Insert
Left
LeftAlt
LeftControl
LeftShift
LeftWindows
Multiply
NumLock
NumPad0 - NumPad9
PageDown
PageUp
PrintScreen
Right
RightAlt
RightControl
RightShift
RightWindows
Scroll
Space
Subtract
Tab
Up

Näppäimistöllä annettuihin syötteisiin pääsee käsiksi KeyboardState -objektin kautta, joka päivittyy jokaisen pelisilmukan yhteydessä. Näppäimistön nykyinen tila voidaan hakea GetState -metodilla. GetPressedKeys -metodilla voidaan kerralla hakea kaikki alhaalla olevat näppäimet Keys tyyppiseen tauluun. Esimerkki näppäimistön tilan hausta on kuviossa 7.2. Yksittäisen näppäimen tilan voi tarkistaa kutsumalla IsKeyDown -metodia ja antamalla sille para-

metrina haluttu näppäin. Metodi palauttaa boolean arvon näppäimen tilasta. (Cawood & McGee 2007, 334-335. ; Hall 2008, 169-171.)

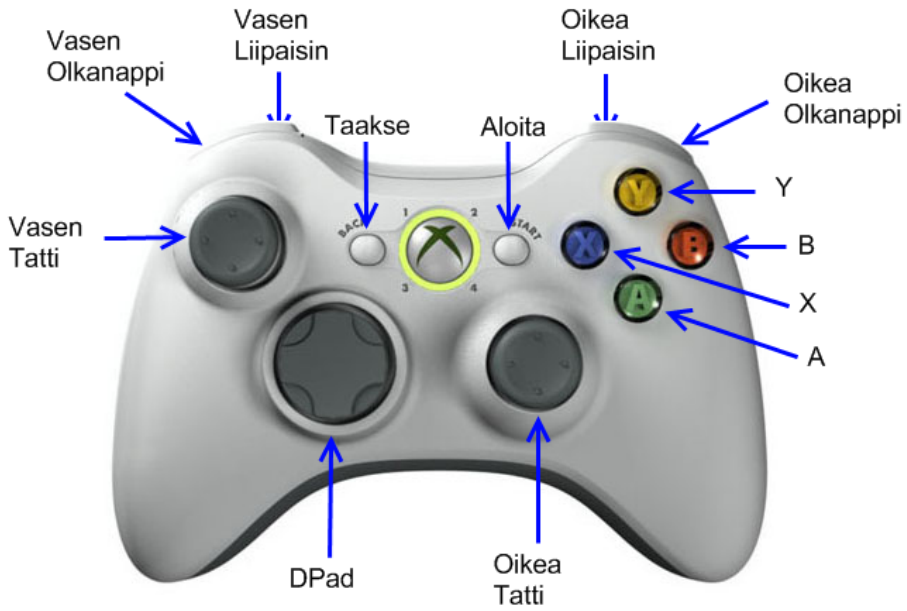
```
//Haetaan viimeisin näppäimistön tila
KeyboardState kbState = Keyboard.GetState();

if (kbState.IsKeyDown(Keys.Up) == true)
{
      //Näppäimistön ylös nuoli on painettu pohjaan
}
```

Kuvio 7.2. Näppäimistön tilan haku

7.3 XBOX 360 -peliohjain

Koska XNA:lla voidaan luoda pelejä sekä PC:lle että XBOX 360:lle on luonnollista, että XNA Framework sisältää tuen XBOX 360 peliohjaimella. Ohjainta voidaan konsolin lisäksi käyttää myös PC:llä. XNA Framework sisältää tuen ohjaimen neljälletoista digitaaliselle näppäimelle, neljälle analogiselle näppäimelle sekä kahdelle vibraattori moottorille. Ohjaimen Media -näppäintä XNA ei kuitenkaan tue. XNA Frameworkin avulla voidaan yhtäaikaisesti ottaa syötteitä vastaan neljästä eri peliohjaimesta. Kuviossa 7.3 on esitetty XBOX 360 peliohjain sekä sen näppäinasettelu. (Cawood & McGee 2007, 2, 335 ; Hall 2008, 150)



Kuvio 7.3. XBOX 360 -peliohjain

Ohjaimen tila voidaan hakea GamePadState -objektiin GetState -metodilla. Kunkin ohjaimen tila täytyy hakea erikseen. Tilan haun jälkeen ennen näppäinten tilojen hakua on suotavaa tarkistaa onko ohjain yhdistetty (Kuvio 7.4).

```
//Haetaan viimeisin 1. ohjaimen tila
GamePadState padPlayer1 = GamePad.GetState(PlayerIndex.One);

if (padPlayer1.IsConnected == true)
{
      //Ohjain yhdistetty ja voidaan hakea näppäinten tilat
}
```

Kuvio 7.4. Ohjaimen tilan tarkistus

Ohjain sisältää neljätoista digitaalista näppäintä, joilla voi olla vain kaksi tilaa: painettu tai vapautettu. Ohjaimen näppäimet ovat seuraavat:

Buttons.A	**Buttons.Start**	**DPad.Down**
Buttons.B	**Buttons.RightShoulder**	**DPad.Left**
Buttons.X	**Buttons.LeftShoulder**	**DPad.Right**
Buttons.Y	**Buttons.RightStick**	**DPad.Up**
Buttons.Back	**Buttons.LeftStick**	**DPad.Down**

Näille kaikille näppäimille löytyy ButtonState attribuutti, joka kertoo näppäimen tilan. Attribuutti on joko Pressed (painettu) tai Released (vapautettu). Kuviossa 7.5 on esimerkki näppäimen tilan tarkistuksesta. (Cawood & McGee 2007, 335-337 ; Hall 2008, 151)

```
//Haetaan viimeisin 1. ohjaimen tila
GamePadState padPlayer1 = GamePad.GetState(PlayerIndex.One);

if (padPlayer1.Buttons.A == ButtonState.Pressed)
{
      //Kyseisen ohjaimen A näppäin on painettuna
}
```

Kuvio 7.5. Näppäimen tila

Digitaalisten näppäinten lisäksi ohjain sisältää analogisia näppäimiä, joita on yhteensä neljä. Analogisten näppäinten ero digitaalisiin on siinä, että tila on jokin arvo niille määrätyllä välillä. Kaksi ensimmäistä ovat liipaisimet. Niiden arvo voi olla liukuluku väliltä 0,0f - 1,0f. Arvo on 0,0f kun liipaisinta ei paineta ja 1,0f kun liipaisin on täysin pohjassa. Kaksi muuta analogista näppäintä ovat vasen ja oikea tatti. Tatilla on y ja x-akseli. Kuviossa 7.6 on esimerkki tattien tilan tarkistuksesta. Molempien akselien arvo voi vaihdella -1,0f ja 1,0f välillä. Esimerkiksi x-akselilla arvo 1,0f kuvaa sitä, että tatti on painettuna täysin oikealla. Jos sekä x-akselin

että y-akselin arvo on 0,0f tarkoittaa tämä sitä, että tattia ei ole liikutettu mihinkään suuntaan. (Cawood & McGee 2007, 337 ; Hall 2008, 151-153.)

```
//Haetaan viimeisin 1. ohjaimen tila
GamePadState padPlayer1 = GamePad.GetState(PlayerIndex.One);

if (padPlayer1.ThumbSticks.Left.X != 0.0f)
{
      //Vasen tatti on joko vasemmalla tai oikealla
}
if (padPlayer1.ThumbSticks.Left.Y != 0.0f)
{
      //Vasen tatti on joko ylhäällä tai alhaalla
}
```

Kuvio 7.6. Analogisten näppäinten tilat

Syötteiden lisäksi ohjaimen kautta on mahdollista lähettää palautetta pelaajalle tärinän muodossa. Tällä tavoin voidaan tehokkaasti edesauttaa ja parantaa pelaajan pelikokemusta. XBOX 360 peliohjain sisältää kaksi erillistä vibraattorimoottoria tähän tarkoitukseen. Molempia moottoreita voidaan säätää erikseen. Moottorit voidaan sammuttaa, käynnistää tai vaihtaa tärinän voimakkuutta SetVibration -metodilla (Kuvio 7.7). Metodi ottaa kolme parametria: ohjaimen tunnisteen, vasemman moottorin voimakkuuden ja oikean moottorin voimakkuuden. Voimakkuudet määrätään liukuluvulla, jonka arvo voi vaihdella välillä 0,0f ja 1,0f. (Cawood & McGee 2007, 338 ; Hall 2008, 153)

```
//Haetaan viimeisin 1. ohjaimen tila
GamePadState padPlayer1 = GamePad.GetState(PlayerIndex.One);

if (padPlayer1.Buttons.A == ButtonState.Pressed)
{
      //Jos ohjaimen A-nappi on pohjassa tärinä päälle
      GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);
}
else
{
      //Jos ohjaimen A-nappi ei ole pohjassa tärinä pois päältä
      GamePad.SetVibration(PlayerIndex.One, 0.0f, 0.0f);
}
```

Kuvio 7.7. Vibraattori moottorin käyttö

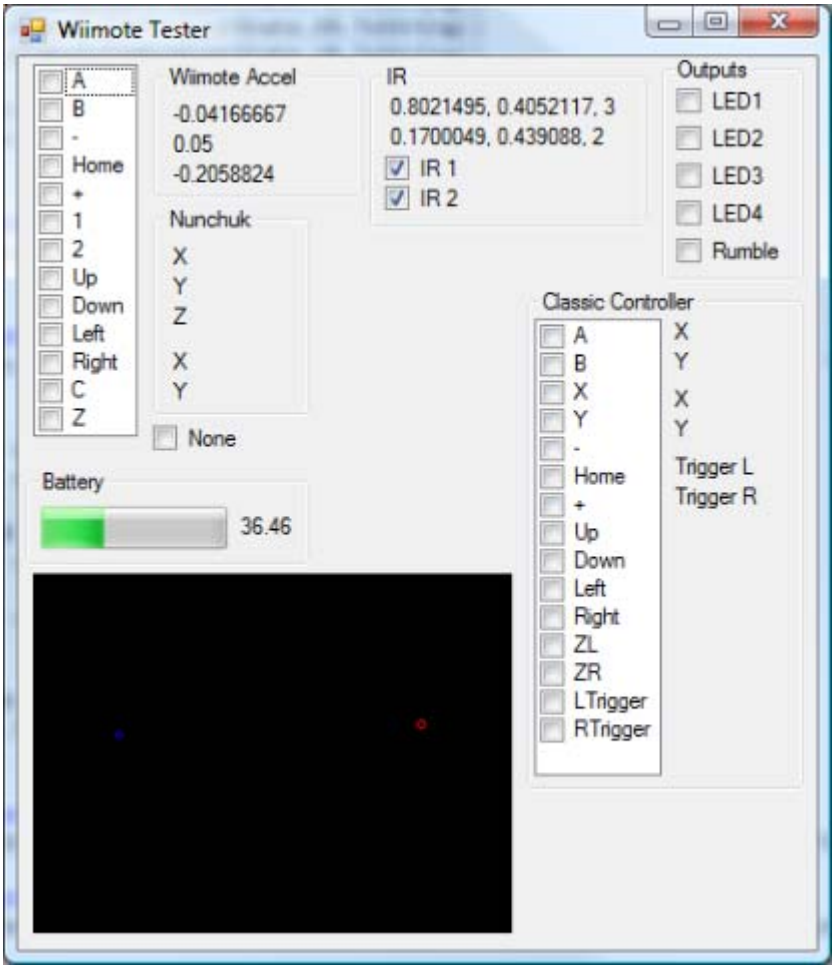
7.4 Wiimote

Wiimote tai Wii Remote (Kuvio 7.8) on Nintendo Wii pelikonsolin ohjain. Wiimote on langaton ja siinä on liiketunnistus sekä infrapunalähetin. Liiketunnistus on kolmelle akselille ja infrapunalähetin on tarkoitettu ”kursorin” ohjausta varten. Wiimoteen voidaan liittää myös Nunchuck lisäohjain, jossa on niin ikään kolmen akselin liikkeentunnistus. Näiden lisäksi Wiimotessa on kaiutin sekä vibraattori toiminto. (Nintendo, 2007)

Kuvio 7.8. Wiimote ja Nunchuck -ohjaimet

XNA Framework ei itsessään tue suoranaisesti muita ohjaimia kuin XBOX 360:n peliohjainta, mutta koska XNA pohjautuu .NET Frameworkiin, sitä voidaan helposti laajentaa muilla kirjastoilla. Yksi esimerkki tästä on WiimoteLib -kirjasto, joka on .NET kirjasto Nintendon Wiimoten hallintaan ja syötteiden vastaanottoon. Wiimote voidaan siis kytkeä PC:hen bluetooth yhteydellä ja WiimoteLib:n avulla saadaan käyttöön kaikki Wiimoten ominaisuudet. (Summala, 2007)

WiimoteLib:iä käyttäen voidaan vastaanottaa kaikki syötteet sekä Wiimote että Nunchuck -ohjaimista. Kaikki nappien painallukset, liikkeet kolmella eri akselilla sekä infrapunapaikannustiedot saadaan pelin käyttöön. Näiden lisäksi saadaan patterin tila, voidaan soittaa ääniä Wiimoten kaiuttimesta, hallita neljää led valoa sekä hallita vibraattorimoottoria. WiimoteLib:n mukana tulee myös testiohjelma, jolla Wiimotea ja sen toimintoja voidaan testata (Kuvio 7.9). Testiohjelma käyttää WiimoteLib -kirjastoa. (Peek, 2007)



Kuvio 7.9. WiimoteLib testiohjelma Wiimote testaukseen (Peek, 2007)

7.5 Äänet

XNA peliin voidaan lisätä ääniä ja musiikkia. Ääniefektit eivät ainoastaan nosta pelin viehätysvoimaa, vaan niitä voidaan myös käyttää pelaajan haasteena. Pelin 3D-äänet mahdollistavat kohteiden paikantamisen pelkän kuuloaistin avulla, esimerkiksi, jos kuulet kaukaisia askeleita oikeassa korvassasi, mutta et voi nähdä ketään kävelemässä, voit tietää että joku kävelee takanasi oikealla. (Cawood & McGee 2007, 386)

7.5.1 XACT

Cross-Platform Audio Creation Tool(XACT) on tehokas äänten tuottamiseen tarkoitettu ympäristö. XACT ei ole työkalu joka on kehitetty vain XNA:ta varten. Se on osa Microsoftin DirectX SDK kirjastoa, joten se pitää sisällään monia ominaisuuksia joihin ei vielä päästä suoraan käsiksi XNA pelistä. Lisää ominaisuuksia tulee koko ajan ja tämä kaventaa ominaisuuksien määrää mitä XNA tukee ja mitä XACT tarjoaa. (Hall 2008, 195.)

XACT luo erillisen Audio projektitiedoston. Tiedosto on lyhenteeltään .xap. Tiedosto pitää sisällään viittaukset wav -ääniin ja niiden esitysasetukset. Tiedosto on käyttökelpoinen sekä Windows alustalle sekä XBOX 360:lle. (Cawood & McGee 2007, 386)

Tiedostoon ei tarvitse koodillisesti viitata missään vaiheessa. Kun .xap tiedostoon luodaan viittaus XNA peliprojektista Visual Studion kautta, peliprojekti osaa automaattisesti luoda globaalin asetustiedoston, wav -äänipankkitiedoston ja tavallisen äänipankkitiedoston, josta tiedostoja voidaan toistaa helposti. Kun viittaus luodaan .xap tiedostoon, se myös osaa automaattisesti luoda viittauksen tiedoston juuripolkuun. (Cawood & McGee 2007, 387)

7.5.2 Audiomoottori

Audiomoottoriluokka alustaa ja ohjaa sisäisiä äänitiedostoja, jotta peliin saadaan äänet. Tämä luokka alustetaan ominaisuuksilla, jotka ladataan globaalista XACT työkalulla luodusta asetustiedostosta. Jos viittaus .xap tiedostoon luodaan Solution Explorerin kautta, globaali asetustiedosto generoidaan automaattisesti projektin suoritus vaiheessa. Vaikka tiedostoa ei fyysisesti voida nähdä samassa kansiossa kuin missä .xap tiedosto sijaitsee, voidaan se silti ladata, jos tiedostopolku on sama kuin projektitiedostolla. Kun luokka on alustettu koodissa, sitä käytetään alustamaan WaveBank ja SoundBank -luokat. Kuten globaaliasetustiedostokin, nämä tiedostot eivät ole fyysisesti samassa audiokansiossa kuin missä .xap tiedosto sijaitsee. Kuitenkin ne voidaan ladata samasta kansiosta kuin missä .xap tiedosto sijaitsee. Kuviossa 7.10 on esitettynä ääniluokkien alustukset. (Cawood & McGee 2007, 387)

```
AudioEngine audioEngine = new AudioEngine(".\\Audio Projekti
Kansio\\GlobalSettings.xgs");

WaveBank waveBank = new WaveBank(audioEngine, ".\\ Audio Projekti
Kansio\\Wave Bank.xwb");


SoundBank soundBack = new SoundBack(audioEngine, ".\\Audio Projekti
Kansio\\Wave Bank.xwb");
```

Kuvio 7.10. Ääniluokkien alustukset.

Wave Bank on kokoelma wave äänitiedostoja, jotka ovat ladattuina ja pakattuina .xwb tiedostossa. SoundBank on kokoelma ohjaustietoja ja cue-tiedostoja, jotka ohjaavat miten ääniä toistetaan pelin sisällä. Cue -tiedostoja käytetään äänten laukaisemiseen SoundBank -luokan sisältä. Cue voi sisältää listan äänistä, jotka soitetaan tietyssä järjestyksessä silloin, kun tietty tapahtuma käynnistyy, tai ne voivat tarjota listan äänistä mistä voidaan valita esimerkiksi satunnainen tai irrallinen tiedosto toistettavaksi. Categories -luokkia käytetään ryhmittämään SoundBank -luokkia, joilla on samankaltaiset ominaisuudet. Esimerkiksi äänet, joilla on sama äänenvoimakkuus, voidaan jakaa yhteen ryhmään, äänet joiden toistaminen täytyy voida keskeyttää tai jatkaa voidaan jakaa toiseen ryhmään.

7.5.3 Äänen esittäminen

Äänten toistamiseen on olemassa kaksi metodia, joiden käyttö on esitettynä kuviossa 7.11. Ensimmäinen sisältää SoundBack -luokan GetCue -metodin käyttämisen äänen hakemiseen ja Play -metodin sen toistamiseen.

```
Cue cue = soundBack.GetCue(String cueName);
cue.Play();
```

Kuvio 7.11. Äänitiedoston hakeminen SoundBank luokasta.

Tämä metodi on käytännöllinen, jos joudutaan esimerkiksi säätämään äänen voimakkuutta tai keskeyttämään ja jatkamaan äänen toistoa. Jos kuitenkin käytetään jatkuvasti GetCue -metodia sekä toistetaan ääni ja poistetaan ääni käyttämällä Dispose -metodia, saattaa ääni katketa toiston aikana. Jos tiedostoja käytetään useaan kertaan, kannattaa IDLE äänet pinota esimerkiksi johonkin kokoelma luokkaan. (Cawood & McGee 2007, 388)

Toinen tapa äänen toistamiseen on käyttää SoundBank -luokan PlayCue -metodia, jonka käyttö esitettynä kuviossa 7.12. Tämä on hyvä tapa toistaa ääniä nopealla tahdilla, kuten esimerkiksi konekiväärin laukaukset. (Cawood & McGee 2007, 388)

```
SoundBank sb = new SoundBank();
sb.PlayCue(CueName);
```

Kuvio 7.12. Äänitiedoston soittaminen.

Tämän metodi ei erikseen poista ääntä muistista, joten se auttaa välttämään äänentoistoon liittyviä ongelmia, jotka saattavat johtua roskien kerääjästä. (Cawood & McGee 2007, 388)

7.5.4 3D äänien ohjelmointi

3D Audio säätelee äänen voimakkuutta ja äänen lähdettä sekä sijoittaa ne eri kaiuttimiin. Jokaisen erillisen äänen voimakkuus ja sijainti määritellään kuuntelijan sijainnin ja etäisyyden mukaan. Kuuntelijaobjekti määritellään käyttämällä AudioListener -luokkaa. Tämä luokka määrää mitä kuuntelija voi kuulla ja miten hän sen kuulee. Normaalisti pelissä on vain yksi kuuntelija, ja tämän sijainti liikkuu kameran mukana. Kuuntelija objektin liikuttaminen kameran mukana mahdollistaa äänien sijainnin ja suuntautumisen päivittämisen kuuntelijaan nähden, kun hän liikkuu pelimaailmassa. AudioEmitter -luokka sisältää äänilähteen sijainnin, nopeuden ja suunnan. AudioEmitter -luokka tarvitaan jokaista äänilähdettä varten. Sekä kuuntelija että äänentoistajaluokka sisältävät neljä erilaista vektoria. Vektoreita päivitetään kuuntelijalle ja jokaiselle äänentoistajalle jokaisen ruudunpäivityksen aikana. Tämän jälkeen eri äänien paikat sekä äänenvoimakkuudet lasketaan Apply3D -metodilla, joka on esitettynä kuviossa 7.13. (Cawood & McGee 2007, 389)

```
Vector3 Forward;  // Suunta
Vector3 Speed;    // Suunta ja voimakkuus
Vector3 Position; // Sijainti
Vector3 Up;       // Suunta ylöspäin, yläviistoon

void Cue.Apply3D(AudioListener listener, AudioEmitter emitter);
```

Kuvio 7.13. 3D-äänien toteuttaminen.

8 PELIPROJEKTI

Opinnäytetyön käytännön osuudessa toteutettiin yksinkertainen ammuskelupeli, jossa pelaaja ampuu kolmeulotteisessa pelimaailmassa erilaisia maalitauluja. Peliin päätettiin tuoda tavallisuudesta poikkeavaa vaihtelua lisäämällä pelin ohjainvaihtoehdoksi Nintendon Wiimote-ohjain.

Peliprojekti toteutettiin projektiluontoisesti ja se tehtiin käytännössä yhden viikon aikana. Peli itsessään on vain teknologiademo, mutta se osoittaa miten XNA helpottaa pelien kehittämistä. Kenelläkään opinnäytetyön tekijöistä ei ollut aikaisempaa kokemusta peliprojekteista tai kokemus oli erittäin vähäistä. Projekti onnistui kaiken kaikkiaan erittäin hyvin ja se tuki hyvin opiskeltua aiheen teoriaa. Työ jaettiin toiminnallisiin kokonaisuuksiin, jotka jaettiin toteutettavaksi eri projektijäsenille. Tässä osuudessa tarkastellaan miten pelin erilaiset tekniset kokonaisuudet ja sisältö toteutettiin.

8.1 Pelilogiikka

Pelilogiikka on verrattain yksinkertainen. Pelin tarkoituksena on ampua ja tuhota mahdollisimman monta tölkkiä ja maalitaulua yhden minuutin aikana. Tähtäimen ohjaukseen voidaan käyttää joko tietokoneen hiirtä tai erillistä Wiimote -ohjainta. Tölkeistä pelaaja saa tietyn määrän pisteitä ja maalitauluista eri määrän pisteitä. Tölkkejä on peliruudulla vain rajallinen määrä ja ne eivät ilmesty takaisin, kun taas maalitaulut laskevat ja nousevat osumien välillä. Kun peliaika loppuu, pelaajalle näytetään hänen saamansa pisteet ja kysytään haluaako hän aloittaa uuden pelin.

Peli sisältää staattisen Player -luokan, jonka kautta pelaajan eri tietoihin päästään käsiksi. Luokka sisältää mm. pelaajan pistemäärän ja pelaajan aseen ammusmäärän. Lisäksi luokassa määritellään miten kauan aseen lataaminen ja ampuminen kestävät. Player -luokasta saadaan myös tietoja siitä, minkälaisessa pelitilassa pelaaja on.

8.2 Pelimaailman luonti

Pelimaailman objektit ja mallit ovat toteutettu 3D Studio Max ohjelmistolla. Mallien tiedostomuotona on käytetty .fbx formaattia, jota XNA tukee suoraan Autodesk FBX Content Importer muodossa. Mallit lisätään peliprojektin Content kansioon ja ladataan koodissa XNA:n tarjoamaan Model -luokkaan (Kuviossa 8.1).

```
_modelBarrel = content.Load<Model>("Barrel");
_modelCan = content.Load<Model>("Can");
_modelFence = content.Load<Model>("Fence");
```

Kuvio 8.1. Mallien lataus.

Ennen objektien latausta ruudulle matriisit on syytä alustaa. Alustuksen avulla objektit saadaan näkymään ruudulle. Tarvittavat matriisit ovat maailmamatriisi, näkymämatriisi ja projektiomatriisi. Näin saadaan maailmaan luotua ns. kamera, joka kuvaa sen mitä pelaaja näkee. Tämän näkymän eteen voidaan sitten asettaa erilaiset pelilliset objektit kuten tynnyrit, maalitaulut jne. Matriisit alustetaan peliprojektissa seuraavasti kuvion 8.2 mukaisesti.

```
protected void Init_WVP()
{
      _graphicsDevice = this.ScreenManager.GraphicsDevice;

      // Calculate the camera matrices.
      Viewport viewport = _graphicsDevice.Viewport;

      float aspectRatio = (float)viewport.Width /
      (float)viewport.Height;

      _mView = Matrix.CreateLookAt(new Vector3(20.0f, 1.0f, 0),
                                   Vector3.Zero,
                                   Vector3.Up);

      _mProjection =
      Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
      aspectRatio, 0.1f, 100);
}
```

Kuvio 8.2. Matriisien alustus.

Tämän jälkeen peliin ladataan vielä kaksi erillistä varjostinta joita voidaan käyttää 3D grafiikan piirtämisen tehosteena. Varjostimet alustetaan seuraavasti: (Kuvio 8.3)

```
protected void Init_Shaders()
{

      _EffectBasicShader = content.Load<Effect>(@"BasicShader");
      _EffectBasicShaderWVPParam =
      _EffectBasicShader.Parameters["fx_WVP"];
      _EffectBasicShader.CommitChanges();

      _EffectTextureShader = content.Load<Effect>("TextureShader");
      _EffectTextureShaderWVPParam =
      _EffectTextureShader.Parameters["fx_WVP"];
      _EffectTextureShaderTextureParam =
      _EffectTextureShader.Parameters["fx_Texture"];
      _EffectTextureShader.CommitChanges();

      this.ScreenManager.GraphicsDevice.RenderState.CullMode =
      CullMode.None;

      _VertPosColor = new
      VertexDeclaration(this.ScreenManager.GraphicsDevice,
      VertexPositionColor.VertexElements);

      _VertPosColorTex = new
      VertexDeclaration(this.ScreenManager.GraphicsDevice,
      VertexPositionColorTexture.VertexElements);
}
```

Kuvio 8.3. Varjostimien alustus.

Kun matriisit ja varjostimet ovat alustettu, voidaan ruudulle piirtää itse pelilogiikkaan liittyviä objekteja. Pelilliset objektit ovat sijoitettuna valmiiksi määritetyille ns. ”kova koodatuille” paikoille. Objekteille on määritetty luokka TargetModelObject, joka pitää sisällään mallin tyypin sekä sen paikkatiedot. Objekteja on sekä staattisia, jotka eivät reagoi osumiin että maalitauluobjekteja, joista pelaaja saa pisteitä. Maalitauluiksi tarkoitettuja malleja säilötään geneerisessä List kokoelma -luokassa. Tätä listaa käytetään myöhemmin hyväksi maalitaulu objektien piirrossa ja osuman tarkistuksen yhteydessä, jolloin kaikki maalitaulu objektit voidaan iteroida helposti läpi. Uusi maalitauluobjekti määritellään ja lisätään generiseen kokoelmaan seuraavasti: (Kuvio 8.4)

```
XNA_WesternShooter.TargetModelObject t1 = new
XNA_WesternShooter.TargetModelObject();
t1.TargetModelType = XNA_WesternShooter.TargetModelType.Jantteri;
t1.TargetTransforms = Matrix.Identity *
                      Matrix.CreateScale(0.021f) *
                      Matrix.CreateRotationY(MathHelper.ToRadians(100)*
                      Matrix.CreateTranslation(13.7f, 0.0f, 2.5f);
_lTargetObjects.Add(t1);
```

Kuvio 8.4. Maalitaulu objektin alustus.

Pelin piirtologiikassa käydään geneerinen lista läpi ja piirretään jokainen siitä löytyvä maalitauluobjekti kuvion 8.5 mukaisesti.

```
case XNA_WesternShooter.TargetModelType.Can:
{
      // Look up the bone transform matrices.
      Matrix[] transforms = new Matrix[_modelCan.Bones.Count];


      _modelCan.CopyAbsoluteBoneTransformsTo(transforms);

      // Draw the model.
      foreach (ModelMesh mesh in _modelCan.Meshes)
      {
            foreach (Effect effect in mesh.Effects)
            {
            Matrix mWorld = transforms[mesh.ParentBone.Index] *
            _lTargetObjects[ii].TargetTransforms;


            effect.Parameters["World"].SetValue(mWorld);

            effect.Parameters["View"].SetValue(_mView);

            effect.Parameters["Projection"].SetValue(_mProjection);
            }

      mesh.Draw();
      }
}
```

Kuvio 8.5. Maalitaulu objektin piirto.

Maalitaulujen lisäksi tarvitaan objekteja, jotka luovat maailman niiden ympärille. Tälläisiä objekteja ovat esimerkiksi maa, tynnyrit ja aita, jotka luovat vaihtelua pelinäkymään. Myös nämä objektit asetetaan ennalta määrätyille paikoilleen pelin sisällön latausvaiheessa. Seuraavassa kuviossa 8.6 on esitetty maan initalisointilogiikan koodi, jossa maa asetetaan pelaajan näkyville. Kuviossa 8.7 nähdään pelimaailma piirrettynä.

```
Vector2 uv = new Vector2(0.0f, 0.0f);
Vector3 pos = new Vector3(0.0f, 0.0f, 0.0f);
Color color = Color.White;


uv.X = 1; uv.Y = 1; pos.X = 20; pos.Y = 0; pos.Z = 15;
_VertGround[0] = new VertexPositionColorTexture(pos, color, uv);

uv.X = 1; uv.Y = 0; pos.X = 0; pos.Y = 0; pos.Z = 15;
_VertGround[1] = new VertexPositionColorTexture(pos, color, uv);

uv.X = 0; uv.Y = 1; pos.X = 20; pos.Y = 0; pos.Z = -15;
VertGround[2] = new VertexPositionColorTexture(pos, color, uv);

uv.X = 0; uv.Y = 0; pos.X = 0; pos.Y = 0; pos.Z = -15;
_VertGround[3] = new VertexPositionColorTexture(pos, color, uv);


_EffectTextureShader.Begin();

_EffectTextureShader.Techniques[0].Passes[0].Begin();

// 1: declare matrices
Matrix matIdentity, matTransl;

// 2: initialize matrices
matIdentity = Matrix.Identity; // always start with identity matrix
matTransl = Matrix.CreateTranslation(0.0f, 0.0f, 0.0f);

// 3: build cumulative world matrix using I.S.R.O.T. sequence
// identity, scale, rotate, orbit(translate & rotate), translate
_mWorld = matIdentity * matTransl;

// 4: pass wvp matrix to shader
_EffectTextureShaderWVPParam.SetValue(_mWorld * _mView * _mProjection);
_EffectTextureShaderTextureParam.SetValue(_Tex2DGrass);
_EffectTextureShader.CommitChanges();

// 5: draw object - select vertex type, primitive type, # of primitives
this.ScreenManager.GraphicsDevice.VertexDeclaration = _VertPosColorTex;

this.ScreenManager.GraphicsDevice.DrawUserPrimitives<VertexPositionColo
rTexture>(PrimitiveType.TriangleStrip, _VertGround, 0, 2);

_EffectTextureShader.Techniques[0].Passes[0].End();
_EffectTextureShader.End();
```

Kuvio 8.6. Staattisten esineiden alustus ja piirto.

Kuvio 8.7. Pelimaailma.

8.3 Syötteet ja osumien tarkistus

Pelaajan antamat syötteet tarkistetaan pelin HandleInput -metodissa, jota kutsutaan vain silloin, kun peli on aktiivisena. Metodissa tarkistetaan pelaajan hiiren tai Wiimote -ohjaimen liikkeet riippuen siitä kumpi ohjain on valittuna. Myös pelaajan tilaa päivitetään jokaisen Update -kutsun aikana. Tällöin huomiota kiinnitetään lähinnä pelaajan aseen tilaan, esim. onko pelaaja juuri ampunut, jolloin hän ei voi heti ampua uudestaan, vai onko pelaaja lataamassa asettaan tai onko pelaajan ase tyhjä. Update -kutsun aikana päivitetään käytettävän peliohjaimen paikkatiedot X ja Y koordinaatistolta globaaleihin muuttujiin, joiden avulla voidaan piirtää pelaajan tähtäin oikeaan kohtaan ruutua.

Maalitaulujen osumat lasketaan jokaisen Update -metodin aikana. Osumat tarkistetaan luomalla viiva tai säde pelaajan näkökentästä kohtisuoraan pelialueen ”takaseinää” kohti. Pisteen

aloituspiste otetaan pelaajan kamerasta sekä osoitinlaitteen X ja Y koordinaatista. Säteen laskeminen on toteutettu kuviossa 8.8.

```
public Ray CalculateCursorRay(Matrix projectionMatrix, Matrix
viewMatrix)
{

Vector2 Position = new Vector2(_fCrosshairX, _fCrosshairY);

// create 2 positions in screenspace using the cursor position. 0 is as
// close as possible to the camera, 1 is as far away as possible.
Vector3 nearSource = new Vector3(Position, 0f);
Vector3 farSource = new Vector3(Position, 1f);

// use Viewport.Unproject to tell what those two screen space positions
// would be in world space. we'll need the projection matrix and view
// matrix, which we have saved as member variables. We also need a //
world matrix, which can just be identity.

Vector3 nearPoint =
ScreenManager.GraphicsDevice.Viewport.Unproject(nearSource,
projectionMatrix, viewMatrix, Matrix.Identity);

Vector3 farPoint =
ScreenManager.GraphicsDevice.Viewport.Unproject(farSource,
projectionMatrix, viewMatrix, Matrix.Identity);

// find the direction vector that goes from the nearPoint to the
farPoint
// and normalize it....
Vector3 direction = farPoint - nearPoint;
direction.Normalize();

// and then create a new ray using nearPoint as the source.
return new Ray(nearPoint, direction);

}
```

Kuvio 8.8. Osuma säteen piirtäminen

Tämän jälkeen voidaan erillisessä metodissa tarkastaa osuuko laskettu säde mihinkään pelin maalitaulumalleista seuraavasti: (Kuvio 8.9)

```
//Get cursor ray
Ray rayCursor = CalculateCursorRay(_mProjection, _mView);

//Loop each target object and set IsHit property
for (int ii = 0; ii < _lTargetObjects.Count; ii++)
{
      //Get "hitbox"
      BoundingSphere bs = new BoundingSphere();
      switch (_lTargetObjects[ii].TargetModelType)
      {
            case TargetModelType.Can:
```

```
                bs = (BoundingSphere)_modelCan.Tag;
                break;

                case TargetModelType.Jantteri:
                bs = (BoundingSphere)_modelJantteri.Tag;
                break;
                }


                //Check that "hitbox" is set
                if (bs != null)
                {

                //Transform "hitbox" to match current target object
                bs = bs.Transform(_lTargetObjects[ii].TargetTransforms);

                //Check hit
                if (bs.Intersects(rayCursor) != null)
                      _lTargetObjects[ii].IsHit = true;
                else
                      _lTargetObjects[ii].IsHit = false;
                }
                else
                      throw new Exception("Model tag missing or correct
                      model wasnt found!")
}
```

Kuvio 8.9. Osumatarkistuksen logiikka

8.4 Pelitilan hallinta

Pelitilan hallintaan ja pelin erilaisten valikoiden toteuttamiseen käytettiin ScreenManager -luokkaa. ScreenManager -luokkaan voidaan syöttää yksittäisiä MenuScreen -luokkia, jotka toimivat pelin eri valikkoruutuina ja joista yksi toimii erillisenä peliruutuna. ScreenManager hallitsee erilaiset siirtymät eri ruutujen välillä ja osaa mm. käyttää pieniä animointeja tehostamaan valikoiden välillä selailua. ScreenManager saa konstruktorissa parametrina pelin Game -luokan, jolloin ScreenManagerin kautta päästään käsiksi erimerkiksi grafiikkalaitteistoon. Kun pelin eri ruudut on saatu toteutettua ja ne lisätään ScreenManager -luokkaan, voidaan niiden välillä liikkua tämän jälkeen helposti. Peli sisältää myös erillisen LoadingScreen -luokan, joka periytyy myös MenuScreen -luokasta. Kuviossa 8.10 nähdään esimerkki kuinka voidaan siirtyä pelin päävalikosta itse peliruutuun.

```
LoadingScreen.Load(ScreenManager, true, new GameplayScreen());
```

Kuvio 8.10. Liikkuminen päävalikosta peliruutuun.

LoadingScreen saa parametreinä ScreenManager -luokan ilmentymän, johon seuraava ladattava ruutu sisältyy sekä itse ladattavan ruudun, joka luodaan suoraan konstruktorin avulla. Lisäksi Load -metodi saa erillisen boolean arvon, jolla kerrotaan onko odotettavissa latausaika vai onko seuraavaksi tuleva ruutu tavallinen valikkoruutu.

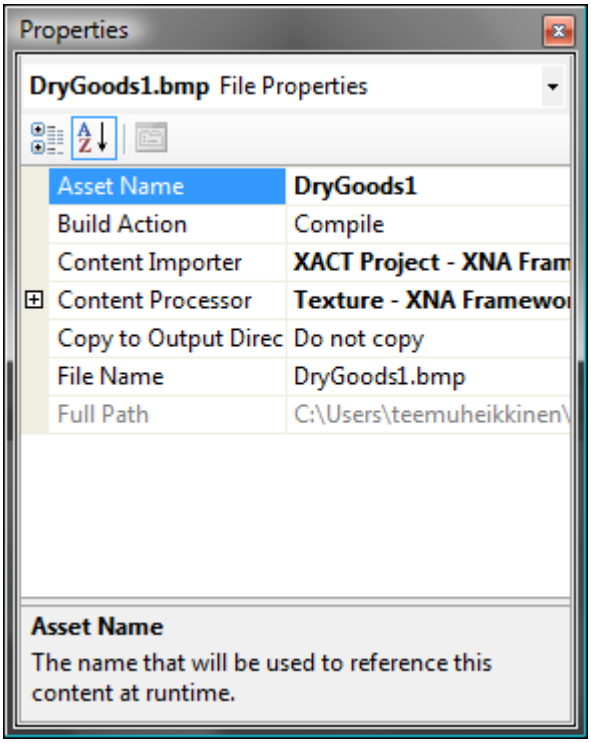
MenuScreen -luokat toimivat käytännössä erillisinä kerroksina, joista jokainen sisältää omat Update ja Draw -metodit. MenuScreen -luokkien erilliset tekstit ja valinnat toteutetaan yksinkertaisella MenuEntry -luokalla. MenuEntry -luokkaan voidaan antaa parametrina teksti jonka se esittää oman Draw -metodinsa aikana. Lisäksi luokalle voidaan antaa yksinkertaisia tapahtumakäsittelijöitä, jotka herätetään erilaisten tapahtumien yhteydessä. Kuviossa 8.11 nähdään pelin aloitusruutu joka on toteutettu MenuScreen ja MenuEntry -luokkien avulla.

Kuvio 8.11. Pelin alkuvalikko.

Peli on toteutettu siten, että esimerkiksi peliaika ei kulu ellei peli-ikkuna ole aktiivinen. Lisäksi peli voidaan keskeyttää painamalla näppäimistön ESC -näppäintä, jolloin ruudulle tuodaan erillinen pysäytysruutu.

8.5 2D -grafiikan toteutus

Peliruudussa pelaaja voi nähdä oman pistemääränsä, ammusmääränsä sekä jäljellä olevan peliajan. Peliruudun grafiikat ovat toteutettu yksinkertaisilla Sprite-grafiikoilla. XNA:n yhteisösivusto Creators XNA (http://creators.xna.com) tarjoaa kotisivuillaan erillisen työkalun, jonka avulla voidaan muuttaa tavallinen Windows TrueType -tyypin fontti sprite kuvatiedostoksi, jolla voidaan esittää tekstiä pelin sisällä. Kuvatiedoston Content Importer ominaisuudeksi valitaan "XACT Project - XNA Framework" ja Content Processor ominaisuudeksi "Texture - XNA Framework". Tämän jälkeen kuvatiedosto voidaan ladata helposti SpriteFont -luokkaan. Kuviossa 8.12 voidaan nähdä DryGoods -fontin asetukset.



Kuvio 8.12. Fontin Import asetukset.

Seuraavaksi voidaan toteuttaa erillinen metodi, jossa pelin HUD -elementit piirretään. Piirtoon käytetään yksinkertaista SpriteBatch -luokkaa, jonka DrawString -metodilla kirjoitetaan pelin tarvitsemat tekstit ruudulle. DrawString -metodi saa parametreina SpriteFont -luokan, jolla teksti piirretään, piirrettävän tekstin, Vector2 -luokan, joka sisältää tiedot koordinaateista mihin teksti on tarkoitus piirtää sekä värin millä teksti piirretään ruudulle.

Myös pelin tähtäin on toteutettu käyttämällä edellä mainittua SpriteBatch -luokkaa. Tässä tapauksessa fontin sijasta piirretään erillinen 2D -tekstuuri. Tekstuurille määritetään erillinen väri, joka piirretään läpinäkyvänä pelin toimesta. Yleensä peliohjelmoinnissa läpinäkyvänä värinä käytetään sinipunaista väriä (eng. Magenta). RGB arvoltaan väri on (255,0,255). Myös muita värejä voidaan käyttää, mutta edellä mainitun värin käyttö näyttäisi olevan vakiintunut käytäntö.

Samalla tavalla piirretään myös pelin muut 2D -tekstuurit, kuten päävalikon taustalla näkyvä puutekstuuri. Näille grafiikoille ei ole yksistään tarvetta määritellä erillistä läpinäkyvää väriä, vaan ne piirretään kauttaaltaan koko ruudun kokoiseksi ilman läpinäkyvyyttä.

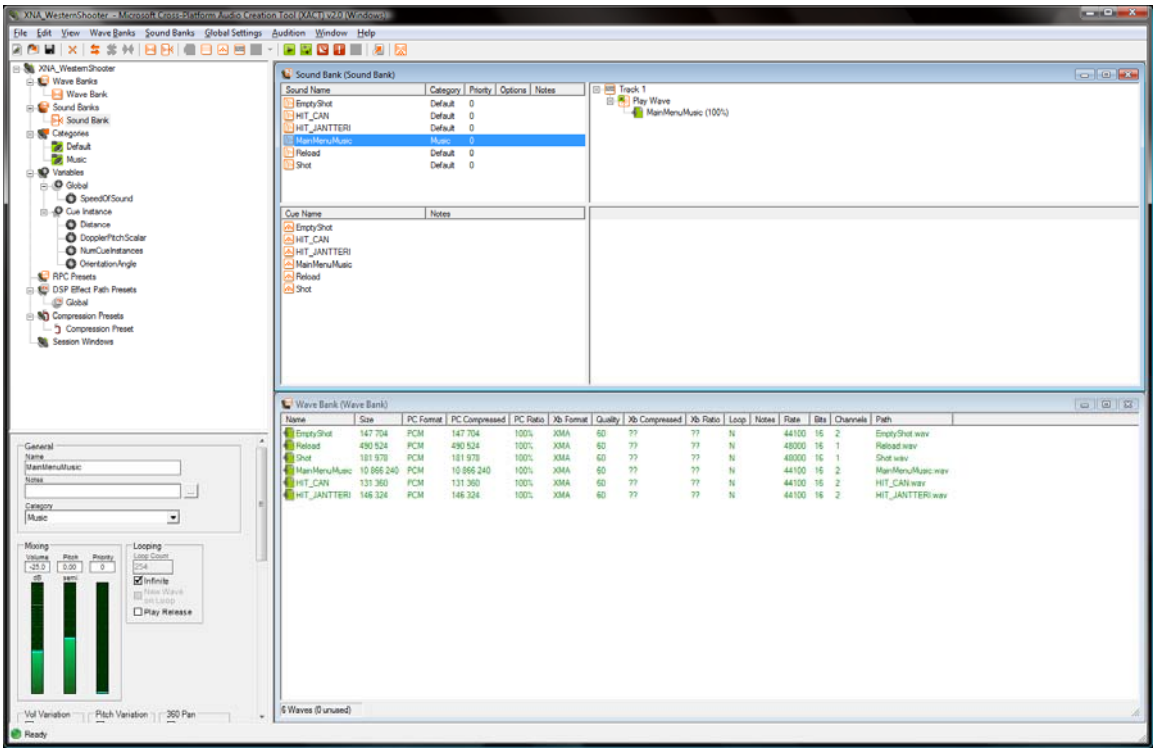
8.6 Äänet

Pelin äänet on toteutettu pitkälti saman kaavan mukaisesti, kuten opinnäytetyön teoriaosuudessa esitetään. Peliprojektiin tarvittavat äänet saatiin SoundSnap sivustolta (http://www.soundsnap.com/) joka tarjoaa ilmaisia ääniefektejä sekä taustamusiikkeja. SoundSnap sivuston tarjoamat äänet ovat sivuston käyttäjien tekemiä eivätkä esimerkiksi jostain maksullisesta äänikirjastosta otettuja, näin ollen ne sopivat tekijänoikeudellisesti tarkasteltuna hyvin myös tällaiseen peliprojektiin.

Jotta äänet saadaan tuotua peliin, niistä täytyy rakentaa erillinen äänikirjasto XNA:n asennuksen mukana tulevalla "Microsoft Cross-Platform Audio Creation Tool (XACT)"-ohjelmistolla. XACT työkalu tukee wav-äänitiedostoja ja se on myös hieman liiankin vaateliaas niiden suhteen. Äänitiedostoja muokattiin peliprojektiin ja XACT työkaluun sopivaksi vielä erillisellä Audacity (http://audacity.sourceforge.net/) ohjelmistolla. Tällä työkalulla voidaan editoida ääntä eri tavoilla ennen kuin se siirretään XACT työkaluun.

Kun äänet on saatu haluttuun formaattiin, siirretään ne ensin XACT työkalun Wave Bank osioon. Kun äänet näkyvät Wave Bank osiossa voidaan ne siirtää Sound Bank osaan, jolloin äänet saadaan sellaiseen muotoon, että niistä voidaan tehdä ilmentymiä myös pelikoodin puolella. Sound Bank osiossa äänille voidaan säätää erillisiä ominaisuuksia, kuten äänenvoimakkuuksia ja kuinka monesti ne soitetaan peräkkäin. Joihinkin ominaisuuksiin voidaan päästä käsiksi myös itse pelikoodista. Näin esimerkiksi voidaan toteuttaa esimerkiksi auton

kaasuttamisesta aiheutuva kiihdytysääni muuttamalla äänen sävelkorkeutta. Kuviossa 8.13 on esitetty pelin ääniprojekti, joka on avoinna XACT -ohjelmassa.



Kuvio 8.13. XACT Projekti.

Seuraavaksi siirretään äänitiedostot peliprojektin Content -kansioon, mutta näitä ei tarvitse lisätä projektiin Visual Studiossa. Sen sijaan Visual Studiossa lisätään Content -kansioon XACT projektin .xap tiedosto. Tämän jälkeen voidaan äänikirjasto ottaa käyttöön koodissa.

Peliprojektiin rakennettiin erillinen SoundPlayer -luokka, jonka avulla ääniä voidaan soittaa tarpeen mukaan. Luokka on määritelty staattiseksi ja sen sisältämät metodit ovat myös staattisia. Näin olleen luokkaan päästään käsiksi eripuolilta projektin koodia. Peliprojektin ääniluokat alustetaan kuvion 8.14 mukaisesti.

```
engine = new AudioEngine("Content\\XNA_WesternShooter.xgs");
soundBank = new SoundBank(engine, "Content\\Sound Bank.xsb");
waveBank = new WaveBank(engine, "Content\\Wave Bank.xwb");
```

Kuvio 8.14. Ääniluokkien alustus.

Sound Bank ja Wave Bank tiedostojen nimet pitäisi olla samoja kuin miten ne ovat määriteltynä XACT projektissa. Mikäli tiedostoja ei kuitenkaan löydy, voi niiden nimet tarkastaa

XNA projektin Build -kansiosta, johon Visual Studio ne kopioi onnistuneen käännöksen jälkeen.

Kun Sound Bank on saatu alustettua, voidaan sen avulla soittaa ääniä kutsumalla PlayCue -metodia. Luokasta voidaan irrottaa äänitiedostoja myös erillisiin Cue -luokkiin. Tällöin niiden soittamista voidaan hallita myös monipuolisemmin, kuten esimerkiksi liittämällä niihin Apply3D -metodilla helposti sijaintitieto. Tällöin pelin sisäinen äänimoottori osaa soittaa äänen sijainnin oikein kolmeulotteisessa maailmassa.

9 POHDINTA

Opinnäytetyön teko sujui ilman suurempia ongelmia. Opinnäytetyösuunnitelmissa laaditut aikataulut eivät loppujen lopuksi pitäneet, tämä johtui lähinnä opinnäytetyön tekijöistä. XNA peliohjelmoinnista löytyi hyviä lähteitä, mutta useimmat kirjalliset lähteet vanhentuvat nopeasti uusien XNA versioiden myötä. XNA Creators -sivusto ja Stephen Cawoodin sekä Pat McGee:n kirjoittama teos ”XNA Game Studio Creators Guide” tarjosivat hyviä ja käytännön läheisiä esimerkkejä.

Opinnäytetyön tekijöillä ei ollut aikaisempaa kokemusta pelikehityksestä ja XNA:sta tai kokemus oli hyvin vähäistä. Työn teoriavaiheen kokoamisessa XNA pelikehityksen ominaisuudet tulivat kuitenkin hyvin tutuksi. Teoriaosuuden työstämisessä haastavinta oli ehdottomasti erilaisten peliteknologiaan liittyvien englanninkielisten termien käännökset. Terminologia on hyvin pitkälti vakiintunutta, joten yksittäisille termeille oli hyvin vaikeaa löytää suomenkielistä vastinetta. Tämä vaikeutti huomattavasti lähdetekstin kääntämistä suomenkieliseksi. Käännöksiä jouduttiin muuntelemaan tekstin tarkistusvaiheessa useampaan kertaan ja useille termeille löytyi tarkistusvaiheessa suomenkieliset vastineet. Lisäksi haasteita asetti tekstin yhdisteleminen kolmen opinnäytetyön tekijän kanssa. Jonkinlainen versionhallinta olisi voinut nopeuttaa ja tehostaa tekstinkäsittelyä huomattavasti.

Peliala on kasvanut räjähtävästi viime vuosien aikana. Pelitalot käyttävät huomattavia summia rahaa pelien kehitykseen ja julkaisuun. Kuitenkin valtaosa pelikehityksestä tapahtuu vielä C/C++-kielillä. XNA tuottaisi varmasti ajallisia voittoja pelikehitykseen, mutta varmasti useat pelikehittäjät eivät ole valmiita jättämään vanhoja kieliä suoritustehon takia. Aloittelijoille ja harrastelijoille XNA kuitenkin tarjoaa erittäin hyvän alustan ja siihen pääsee nopeasti sisään.

Opinnäytetyönsuunnitelmassa listattiin kolme keskeistä tavoitetta:

- Selvittää, miten toteutetaan pelimoottori / teknologiademo hyväksikäyttäen Microsoftin XNA teknologiaa.

- Esitellään peliohjelmoinnin eri osa-alueita teoreettisesti ja tutkitaan samalla XNA:n etuja aikaisempiin toteutusmenetelmiin.

- Lisäksi pyritään tuottamaan kattava ohjeistus peliohjelmointiin ja XNA:n käyttöön peliohjelmoinnissa.

Ensimmäisessä tavoitteessa onnistuttiin ja yksinkertainen teknologia demo saatiin luotua. Pelidemossa testattiin teoriaosuuden toimivuutta käytännössä ja opinnäytetyön tekijät saivat käytännön kokemusta pelin tekemisestä. Pelidemo toteutettiin käytännössä yhden lomaviikon aikana. Näin ollen aikataulun kanssa oli jonkin verran kiirettä. Kaikki opinnäytetyön tekijät tekevät ohjelmointityötä päivätyökseen, joten tämä syö myös hieman energiaa muihin projekteihin panostukselta. Peli kuitenkin saatiin toteutettua viikossa pieniä viilailuja lukuun ottamatta. Toisena tavoitteena oli eri peliohjelmoinnin osa-alueiden käsittely teoreettisesti. Tässäkin onnistuttiin hyvin, mutta etuja aikaisempiin toteutusmenetelmiin olisi voitu verrata säännöllisemmin. Opinnäytetyö käy hyvin opetusmateriaaliksi ja sitä tullaankin käyttämään Kajaanin Ammattikorkeakoulun XNA kursseilla. Lisäksi opinnäytetyön esitys toteutetaan osana kurssin oppituntia.

Aiheena XNA oli kaiken kaikkiaan hyvin mielenkiintoinen. Pelien ohjelmointi tuo hyvää vaihtelua tavallisen bisneslogiikkasovellusten tekoon, joita opinnäytetyön tekijät tekevät työnään. Lisäksi pelien pelaaminen on kuulunut tai kuuluu opinnäytetyön tekijöiden jokapäiväiseen elämään, joten on ollut hyvä nähdä miten pelit käytännössä toimivat pinnan alla. Pelkän ohjelmoinnin lisäksi pelin toteutuksen yhteydessä opittiin käyttämään myös muita työkaluja kuten 3D-mallinnus työkaluja sekä äänen käsittelyyn liittyviä ohjelmistoja.

LÄHTEET

Autodesk. 2008. Saatavilla: http://www.autodesk.com/fbx (Luettu 14.10.2008)

Cawood, Stephen. & McGee, Pat. 2007. XNA Game Studio Creators Guide. The McGraw Hill Companies.

Hall, Joseph. 2008. XNA Game Studio Express. Thompson Course Technology.

Mathematics of Lighting, 2008. Saatavilla: http://msdn.microsoft.com/en-us/library/bb147178(VS.85).aspx (Luettu 16.5.2008)

Nintento, 2007 Saatavilla: http://www.nintendo.co.uk/NOE/en_GB/systems/technical_details_1072.html (Luettu 3.4.2008)

Nitschke, Benjamin. 2007. Professional XNA Game Programming: Wiley Publishing

OpenSG, 2005. Saatavilla: http://www.opensg.org/doc-1.6.0/Windows.html (Luettu 16.5.2008)

Peek, Brian. 2007. Saatavilla: http://blogs.msdn.com/coding4fun/archive/2007/03/14/1879033.aspx (Luettu 3.4.2008)

Project BlueStreak 2006. Saatavilla: http://ghost.halomaps.org/bluestreak/animation/ (Luettu 14.10.2008)

Summala, Antti. 2007. Saatavilla: http://www.yougamers.com/news/8465_microsoft_shows_nintendo_a_little_love/ (Luettu 3.4.2008)

Wikipedia a. Saatavilla: http://en.wikipedia.org/wiki/Community_Technology_Preview#Beta (Luettu 16.5.2008)

Wikipedia b. Saatavilla: http://en.wikipedia.org/wiki/Microsoft_XNA (Luettu 16.5.2008)

Wikipedia c. Saatavilla: http://en.wikipedia.org/wiki/Heightmap (Luettu 15.10.2008)

XNA Team Blog 2008. Saatavilla: http://blogs.msdn.com/xna/archive/2008/02/20/announcing-xna-game-studio-3-0-and-zune.aspx (Luettu 23.5.2008)

XNA Creators 2008. Saatavilla: http://creators.xna.com/Headlines/developmentaspx/archive/2007/01/01/Generated-Geometry-Sample.aspx (Luettu 15.10.2008)

LIITTEIDEN LUETTELO

ESIMERKKI PELIPROJEKTI

```
//-------------------------------------------------------------
// Esimerkki Peliprojekti - Program.cs tiedoston sisältö
//-------------------------------------------------------------
using System;

namespace Game
{
    static class Program
    {
        // Pelin käynnistyslohko
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}



//-------------------------------------------------------------
//Esimerkki Peliprojekti - Game.cs tiedoston sisältö
//-------------------------------------------------------------
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Storage;
using NS_Camera;
using System.Runtime.InteropServices;
#endregion

namespace Game
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        //-------------------------------------------------------------
        // Hiiren osoittimen hallintaan tarvittavat määrittelyt
        //-------------------------------------------------------------
      #if !XBOX
        MouseState mMouse;        // Muuttuja hiirelle

        // dll referenssi WIN32 API:n, hiiren paikkatietojen hallintaa
        // varten
        [DllImport("user32.dll")]
        static extern bool SetCursorPos(int X, int Y);
        [DllImport("user32.dll")]
        static extern int GetCursorPos(ref tPoint lpPoint);

        // strukti hiiren paikkatietoja varten
        struct tPoint { public int X, Y; }
```

```
        // Metodi hiiren paikkatietojen hakua varten
        public static Point GetCursorPoint()
        {
            tPoint point;
            point.X = 0; point.Y = 0;
            GetCursorPos(ref point);
            return new Point(point.X, point.Y);
        }
    #endif
        //--------------------------------------------------------------
        // Game luokan jäsenmuuttujat
        //--------------------------------------------------------------

        // Vakiot
        private const float BOUNDARY = 16.0f;

        GraphicsDeviceManager   gfx;
        ContentManager          content;

        // Varjostin  - BasicShader.fx
        private Effect          mfx;

        // kumulatiivinen transformaatio matriisi w*v*p (World, View,
        //Projection, Viittaa varjostimen WVP muuttujaan
        private EffectParameter worldViewProjParam;

        // Varjostin - TexturedShader.fx
        private Effect          mfxTex;

        // kumulatiivinen transformaatio matriisi w*v*p (World, View,
        //Projection, Viittaa varjostimen WVP muuttujaan
        private EffectParameter mfxTex_WVP;

        //Viittaa varjostimen tekstuurimuuttujaan
        private EffectParameter mfxTexture;

        // Kamere muuttujat
        private CCamera         cam = new CCamera();
        private Matrix          mMatWorld;
        private Matrix          mMatView;
        private Matrix          mMatProj;

        // syötelaitteet
        private GamePadState[]  mGamePadState = new GamePadState[4];

        // verteksityypit ja puskurit
        private VertexDeclaration               mVertPosColor;
        private VertexDeclaration               mVertPosColorTex;
        private VertexPositionColorTexture[]    mVertGround = new
VertexPositionColorTexture[4];

        //tekstuurimuuttuja
        private Texture2D                       mTexGrass;
        //Varjostin
        private Effect                          mfxEffect;
        //kumulatiivinen transformaatio matriisi (WVP)
        private EffectParameter                 mfxEffectWVP;
        //Muuttujat sinisen värin viittausta varten
        private EffectParameter                 mfxEffectBlue;
```

```
  //Muuttuja X-position viittausta varten
  private EffectParameter                 mfxEffectPosX;

  private VertexPositionColor[]           mVertices   = new
VertexPositionColor[4];

  private float                           mfBlue          = 0.0f;
  private bool                            mbBlueIncrease  = true;

  private float                           mfXpos          = 0.0f;
  private bool                            mbXIncrease     = true;


  /// <summary>
  /// Alustetaan GraphicsDeviceManager ja
  /// ContentManager objektit piirtoa varten
  /// </summary>
  public Game1()
  {
      gfx     = new GraphicsDeviceManager(this);
      content = new ContentManager(Services);
  }

  void setPositionX(GameTime gameTime)
  {
      //Lisätään tai vähennetään x-akselin arvoa peliajan mukaan
      if (mbXIncrease)
          mfXpos +=
            (float)gameTime.ElapsedGameTime.Milliseconds/1000.0f;
      else
          mfXpos -=
            (float)gameTime.ElapsedGameTime.Milliseconds/1000.0f;

      //Vähennetään kunnes x-akselin arvo on -1
      if (mfXpos <= -1.0f)
          mbXIncrease = true;
      //Lisätään kunnes x-akselin arvo 1
      else if (mfXpos >= 1.0f)
          mbXIncrease = false;

      //Asetetaan uusi x-akselin arvo varjostimelle
      mfxEffectPosX.SetValue(mfXpos);

      //Hyväksyy muutokset varjostimella
      mfxEffect.CommitChanges();
  }
  void setBlueColor(GameTime gameTime)
  {
      //Lisätään tai vähennetään sinisen värin arvoa peliajan
      //mukaan
      if (mbBlueIncrease)
          mfBlue +=
      (float)gameTime.ElapsedGameTime.Milliseconds / 1000.0f;
      else
          mfBlue -=
      (float)gameTime.ElapsedGameTime.Milliseconds / 1000.0f;
```

```
            //Vähennetään jos värin arvo on suurempi kuin 0
            if (mfBlue <= 0.0f)
                mbBlueIncrease = true;
            //Lisätään jos värin arvo on pienempi kuin 1
            else if (mfBlue >= 1.0f)
                mbBlueIncrease = false;

            //Asetetaan uusi väriarvo varjostimelle
            mfxEffectBlue.SetValue(mfBlue);
            //Hyväksy muutokset
            mfxEffect.CommitChanges();
        }

        private void initVertices()
        {
            Vector3 pos = new Vector3(0.0f, 0.0f, 0.0f);
            Color color = Color.White;

            // Alustaa verteksit
            pos.X = -3.0f; pos.Y = 3.0f; pos.Z = 15.0f;

            // top right
            mVertices[0] = new VertexPositionColor(pos, color);

            // bottom right
            pos.X = -3.0f; pos.Y = -3.0f; pos.Z = 15.0f;
            mVertices[1] = new VertexPositionColor(pos, color);

            // top left
            pos.X = 3.0f; pos.Y = 3.0f; pos.Z = 15.0f;
            mVertices[2] = new VertexPositionColor(pos, color);

            //bottom left
            pos.X = 3.0f; pos.Y = -3.0f; pos.Z = 15.0f;
            mVertices[3] = new VertexPositionColor(pos, color);
    }

        /// <summary>
        /// Liikuttaa kameraa eteen- ja taaksepäin
        /// Käyttäjän painaessa ylös- tai alasnäppäintä.
        /// Toimii myös gamepadilla.
        /// </summary>
        float move()
        {
            KeyboardState kbState   = Keyboard.GetState();
            GamePadState  gpState   = getNewState(mGamePadState[0]);
            float         fMove     = 0.0f;
            const float   kScale    = 1.50f;

            //Tarkistetaan käyttääkö käyttäjä gamepadia
            if (mGamePadState[0].IsConnected)
            {
                if (gpState.ThumbSticks.Left.Y != 0.0f)
                    fMove = (kScale * gpState.ThumbSticks.Left.Y);
            }
            else
                //Käyttäjä ei käytä gamepadia, vaan näppäimistöä
                if (kbState.IsKeyDown(Keys.Up))
                    fMove = (1.0f);
                else if (kbState.IsKeyDown(Keys.Down))
```

```
            fMove = (-1.0f);
    return fMove;
}

/// <summary>
/// Liikuttaa kameraa sivuaskelin
/// </summary>
float strafe()
{
    KeyboardState kbState = Keyboard.GetState();
    GamePadState  gpState = getNewState(mGamePadState[0]);

    //Tarkistetaan käyttääkö käyttäjä gamepadia
    if (mGamePadState[0].IsConnected)
    {
        if(gpState.ThumbSticks.Left.X != 0.0f)
            return gpState.ThumbSticks.Left.X;
    }
    //Käyttäjä ei käytä gamepadia, vaan näppäimistöä
    else if (kbState.IsKeyDown(Keys.Left)) // strafe left
        return -1.0f;
    else if (kbState.IsKeyDown(Keys.Right)) // strafe right
        return 1.0f;
    return 0.0f;
}

/// <summary>
/// Palauttaa viimeisimmät tilan gamepadilta
/// </summary>
 GamePadState getNewState(GamePadState state)
{
    return state;
}

/// <summary>
/// Päivittää kaikkien gamepadien tilan
/// </summary>
private void UpdateGamePad()
{
    mGamePadState[0] = GamePad.GetState(PlayerIndex.One);
    mGamePadState[1] = GamePad.GetState(PlayerIndex.Two);
    mGamePadState[2] = GamePad.GetState(PlayerIndex.Three);
    mGamePadState[3] = GamePad.GetState(PlayerIndex.Four);
}

/// <summary>
/// Pelin alustukset
/// </summary>
protected override void Initialize()
{
    // asettaa ikkuna-asetukset, varjostimet, jne
    init_XNA_app();

    // alustetaan verteksityyppit
    mVertPosColor       = new
    VertexDeclaration(gfx.GraphicsDevice,
    VertexPositionColor.VertexElements);

    mVertPosColorTex    = new
    VertexDeclaration(gfx.GraphicsDevice,
```

```
            VertexPositionColorTexture.VertexElements);

            // alustetaan peliobjektit
            init_ground();

#if !XBOX
            SetCursorPos(Window.ClientBounds.Width / 2,
Window.ClientBounds.Height / 2);
#endif
            //Ladataan erillinen varjostin
            mfxEffect       =
            content.Load<Effect>(@"shaders\IntroShader");

            //Asetetaan varjostimen muuttuja viittaukset
            mfxEffectWVP    = mfxEffect.Parameters["fx_WVP"];
            mfxEffectBlue   = mfxEffect.Parameters["fx_Blue"];
            mfxEffectPosX   = mfxEffect.Parameters["fx_PosX"];

            //Alustetaan verteksit
            initVertices();

            // Kaikki omat alustukset tulee suorittaa tämän rivin
            //yläpuolella
            base.Initialize();
        }

        private void drawRectangle()
        {
            // 1: Määritellään matriisit
            Matrix matIdentity, matTransl;
            // 2: Alustetaan matriisit
            matIdentity = Matrix.Identity;

            // 3: rakennetaan kumulatiivinen matriisi käyttäen
            //I.S.R.O.T. sekvenssiä
            // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja
            //siirto
            // (ENG) identity, scale, rotate, orbit(translate &
            //rotate),translate
            matTransl = Matrix.CreateTranslation(0.0f, -0.9f, 0.0f);

            mMatWorld = matIdentity * matTransl;

            // 4: Asetetaan WVP muuttuja varjostimelle
            mfxEffectWVP.SetValue(mMatWorld * mMatView * mMatProj);

            //Hyväksyy muutokset
            mfxEffect.CommitChanges();

            // 5: piirrä objekti – valitse verteksityyppi,
            //primitiivityyppi, piirrettävien verteksien määrä
            gfx.GraphicsDevice.VertexDeclaration = mVertPosColor;
            gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
            PrimitiveType.TriangleStrip, mVertices, 0, 2);
        }

        /// <summary>
        /// Lataa graafinen sisältö (kuvat, tekstuurit, jne)
```

```
  /// </summary>
protected override void LoadGraphicsContent(bool loadAllContent)
{
      if (loadAllContent)
      {
        //Lataa tekstuuri
        mTexGrass = content.Load<Texture2D>(".\\Images\\grass");
      }
}

  /// <summary>
  /// Vapauta ladattu graafinen sisältö
  /// </summary>
protected override void UnloadGraphicsContent(bool
unloadAllContent)
{
      if (unloadAllContent == true)
      {
          //Vapauta sisältö
          content.Unload();
      }
}

  /// <summary>
  /// Pelilogiikan päivitys, virheentarkistukset,
  /// törmäystarkistukset
  /// </summary>
  protected override void Update(GameTime gameTime)
  {
      KeyboardState kbState = Keyboard.GetState();

      // Pelin lopetus logiikka
      if (GamePad.GetState(PlayerIndex.One).Buttons.X ==
      ButtonState.Pressed
          || kbState.IsKeyDown(Keys.Escape))
      { this.Exit();

      // Päivitä kameran aika
      cam.set_frame_interval(gameTime);

      // Päivitä ohjaimet
      UpdateGamePad();
#if !XBOX
      mMouse = Mouse.GetState();
#endif
      cam.move(move());
      cam.strafe(strafe());

      // Päivitä muuttavat matriisit
      set_view_matrix(gameTime);

      setBlueColor(gameTime);
      setPositionX(gameTime);


      base.Update(gameTime);
  }

  /// <summary>
  /// Alustaa verteksit maata varten
```

```
/// </summary>
private void init_ground()
{
    Vector2 uv  = new Vector2(0.0f, 0.0f);
    Vector3 pos = new Vector3(0.0f, 0.0f, 0.0f);
    Color color = Color.White;

    // Alustaa verteksien paikka – ja tekstuuriarvot
    uv.X = 10.0f; uv.Y = 10.0f; pos.X = -BOUNDARY; pos.Y =
    0.0f; pos.Z = -BOUNDARY;

    mVertGround[0] = new VertexPositionColorTexture(pos, color,
    uv); //front right

    uv.X = 10.0f; uv.Y = 0.0f; pos.X = -BOUNDARY; pos.Y = 0.0f;
    pos.Z = BOUNDARY;

    mVertGround[1] = new VertexPositionColorTexture(pos, color,
    uv); //back right

    uv.X = 0.0f; uv.Y = 10.0f; pos.X = BOUNDARY; pos.Y = 0.0f;
    pos.Z = -BOUNDARY;

    mVertGround[2] = new VertexPositionColorTexture(pos, color,
    uv); //front left

    uv.X = 0.0f; uv.Y = 0.0f; pos.X = BOUNDARY; pos.Y = 0.0f;
    pos.Z = BOUNDARY;

    mVertGround[3] = new VertexPositionColorTexture(pos, color,
    uv); //back left
}

/// <summary>
/// Varjostimien ja peli-ikkunan alustus
/// </summary>
private void init_XNA_app()
{
    // ikkunan otsikko
    Window.Title = "Esimerkki Peliprojekti";

    //lataa BasicShader.fx ja aseta WVP muuttujaan viittaus
    mfx = content.Load<Effect>(@"shaders\BasicShader");
    worldViewProjParam = mfx.Parameters["fx_WVP"];
    mfx.CommitChanges();

    //lataa TextureShader.fx ja aseta WVP ja Texture muuttujiin
    //viittaukset
    mfxTex     =
    content.Load<Effect>(@"shaders\TextureShader");
    mfxTex_WVP = mfxTex.Parameters["fx_WVP"];
    mfxTexture = mfxTex.Parameters["fx_Texture"];
    mfxTex.CommitChanges();

    // aseta kamera matriksi
    set_proj_matrix(); // asetetaan vain kerran

    //Otetaan culling mode pois päältä
    gfx.GraphicsDevice.RenderState.CullMode = CullMode.None;
}
```

```
/// <summary>
/// Piirrä teksturoitu maa
/// </summary>
private void draw_ground()
{
    // 1: Määrittele matriisit
    Matrix matIdentity, matTransl;

    // 2: Alusta matriisit
    matIdentity = Matrix.Identity;

    matTransl   = Matrix.CreateTranslation(0.0f, -0.9f, 0.0f);

    // 3: rakennetaan kumulatiivinen matriisi käyttäen
    //I.S.R.O.T. sekvenssiä
    // (FIN) yksikkömatriisi, skaalaus, pyöritys, kierto ja
    //siirto
    // (ENG) identity, scale, rotate, orbit(translate &
    //rotate),translate
    mMatWorld   = matIdentity * matTransl;

    // 4: aseta wvp matriisi varjostimeen
    mfxTex_WVP.SetValue(mMatWorld * mMatView * mMatProj);
    mfxTexture.SetValue(mTexGrass);

    //Hyväksy muutokset
    mfxTex.CommitChanges();

    // 5: piirrä objekti – valitse verteksityyppi,
    //primitiivityyppi, piirrettävien verteksien määrä
    gfx.GraphicsDevice.VertexDeclaration = mVertPosColorTex;

    gfx.GraphicsDevice.DrawUserPrimitives<VertexPositionColorTe
    xture>(PrimitiveType.TriangleStrip, mVertGround, 0, 2);
}

/// <summary>
/// Säädetään näkökenttää gamepadin tai hiiren
/// perusteella
/// </summary>
Vector2 changeView(GameTime gameTime)
{
    const float SENSITIVITY = 250.0f;

    KeyboardState kbState = Keyboard.GetState();

    int iWidthMiddle    = Window.ClientBounds.Width/2;
    int iHeightMiddle   = Window.ClientBounds.Height/2;

    Vector2 v2Change    = new Vector2(0.0f, 0.0f);

    //Tarkistetaan käyttääkö käyttäjä gamepadia
    if (mGamePadState[0].IsConnected == true)
    {
          float kScaleY =
          (float)gameTime.ElapsedGameTime.Milliseconds/50.0f;
          v2Change.Y = kScaleY *
          mGamePadState[0].ThumbSticks.Right.Y * SENSITIVITY;
```

```
            v2Change.X = mGamePadState[0].ThumbSticks.Right.X *
            SENSITIVITY;
        }
        else
        {
              //Käyttäjä käyttää hiirtä

              // Tarkistetaan käyttääkö käyttäjä XBOX:n kanssa
              //hiirtä
        #if !XBOX
            float kScaleY =
        (float)gameTime.ElapsedGameTime.Milliseconds / 100.0f;
            float kScaleX =
        (float)gameTime.ElapsedGameTime.Milliseconds/400.0f;

            // hae kursorin positio
            Point point = GetCursorPoint();
            int iY      = point.Y;
            int iX      = point.X;

            // Muuta näkymää ellei hiiri ole x-akselilla keskellä
            //näyttöä
            if (iX != iWidthMiddle)
            {
                v2Change.X  = iX - iWidthMiddle;
                v2Change.X /= kScaleX;

            }
            // Muuta näkymää ellei hiiri ole y-akselilla keskellä
            //näyttöä
            if (iY != iHeightMiddle)
            {
                v2Change.Y  = iY - iHeightMiddle;
                v2Change.Y /= kScaleY;
            }

            // aseta kursori keskelle näyttöä
            SetCursorPos(iWidthMiddle, iHeightMiddle);
        #endif
        }
        return v2Change;
    }

    /// <summary>
    /// Aseta projektiomatriisi
    /// </summary>
    void set_proj_matrix()
    {
        //Alusta projektiomatriisi
        mMatProj =
        Matrix.CreatePerspectiveFieldOfView((float)Math.PI/4.0f,
        (float)Window.ClientBounds.Width/(float)Window.ClientBounds
        .Height, 0.005f, 1000.0f);
    }

    /// <summary>
    /// Määritellään objektien sijanti suhteessa kameraan
    /// </summary>
    void set_view_matrix(GameTime gameTime)
    {
```

```
            Vector2 v2View = changeView(gameTime);
            cam.changeView(v2View.X, v2View.Y);
            mMatView = Matrix.CreateLookAt(cam.m_vPos, cam.m_vView,
            cam.m_vUp);
        }

        /// <summary>
        /// Pelin piirtometodi
        /// </summary>
        protected override void Draw(GameTime gameTime)
        {
            // Tyhjennä näyttö (Täytä määrätyllä värillä)
            gfx.GraphicsDevice.Clear(Color.Black);

            // aloita varjostimen käyttö - IntroShader.fx
            mfxEffect.Begin();
            mfxEffect.Techniques[0].Passes[0].Begin();
                // piirrä objektit
                drawRectangle();
            // lopeta varjostimen - IntroShader.fx
            mfxEffect.Techniques[0].Passes[0].End();
            mfxEffect.End();

            // lopeta piirto
            base.Draw(gameTime);
        }
    }
}



//-----------------------------------------------------------------
// Esimerkki Peliprojekti - CCamera.cs tiedoston sisältö
//-----------------------------------------------------------------
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace NS_Camera
{
    public class CCamera
    {
        public  Vector3 m_vPos, m_vUp, m_vView;
        private float  mfTimeLapse;

        /// <summary>
        /// Konstruktori
        /// </summary>
        public CCamera()
        {

            mfTimeLapse = 0;

            m_vPos      = new Vector3(0.0f, 0.0f, 0.0f);

            m_vView.X   = 0.0f;
            m_vView.Y   = 0.0f;
            m_vView.Z   = 0.5f;

            m_vUp.X     = 0.0f;
```

```
    m_vUp.Y     = 1.0f;
    m_vUp.Z     = 0.0f;
}

/// <summary>
/// Kierto akselilla määrätyn määrän asteina
/// </summary>
private Vector4 get_rotation_quaternion(float f_deg, Vector3
vAxisA)
{
    Vector4 vAxisUnit;
    Vector4 vAxis = new Vector4(vAxisA.X, vAxisA.Y, vAxisA.Z,
    0.0f);

    // normalisoi tarvittaessa
    if ((vAxis.X != 0 && vAxis.X != 1) ||
        (vAxis.Y != 0 && vAxis.Y != 1) ||
        (vAxis.Z != 0 && vAxis.Z != 1))
    {
        vAxisUnit = Vector4.Normalize(vAxis);
    }

    float f_angle = f_deg * (float)Math.PI / 180.0f;
    float f_sin = (float)Math.Sin(f_angle / 2.0f);

    // luo kvaternio
    Vector4 vQT = new Vector4(0.0f, 0.0f, 0.0f, 0.0f);
    vQT.X = vAxis.X * f_sin;
    vQT.Y = vAxis.Y * f_sin;
    vQT.Z = vAxis.Z * f_sin;
    vQT.W = (float)Math.Cos(f_angle / 2.0f);

    Vector4 vQTUnit;
    vQTUnit = Vector4.Normalize(vQT);

    return vQTUnit;
}

/// <summary>
/// Viimeistele muutokset
/// </summary>
private void update_camera_view(float f_angle, Vector3
 vDirection)
{
    Vector4 vLookQT;
    Vector4 vQT;

    // luo kvaternio määrätylle akselille
    vQT = get_rotation_quaternion(f_angle, vDirection);


    vLookQT.X = m_vView.X - m_vPos.X;
    vLookQT.Y = m_vView.Y - m_vPos.Y;
    vLookQT.Z = m_vView.Z - m_vPos.Z;
    vLookQT.W = 0;

    Vector4 vConj = new Vector4(-vQT.X, -vQT.Y, -vQT.Z, vQT.W);

    Vector4 vQuat;
```

```
        vQuat.X = vQT.W * vLookQT.X + vQT.X * vLookQT.W + vQT.Y *
        vLookQT.Z - vQT.Z * vLookQT.Y;
        vQuat.Y = vQT.W * vLookQT.Y - vQT.X * vLookQT.Z + vQT.Y *
        vLookQT.W + vQT.Z * vLookQT.X;
        vQuat.Z = vQT.W * vLookQT.Z + vQT.X * vLookQT.Y - vQT.Y *
        vLookQT.X + vQT.Z * vLookQT.W;
        vQuat.W = vQT.W * vLookQT.W - vQT.X * vLookQT.X - vQT.Y *
        vLookQT.Y - vQT.Z * vLookQT.Z;

        // valmistele kvaternio
        Vector4 qNewView;
        qNewView.X = vQuat.W * vConj.X + vQuat.X * vConj.W +
        vQuat.Y * vConj.Z - vQuat.Z * vConj.Y;
        qNewView.Y = vQuat.W * vConj.Y - vQuat.X * vConj.Z +
        vQuat.Y * vConj.W + vQuat.Z * vConj.X;
        qNewView.Z = vQuat.W * vConj.Z + vQuat.X * vConj.Y -
        vQuat.Y * vConj.X + vQuat.Z * vConj.W;
        qNewView.W = vQuat.W * vConj.W - vQuat.X * vConj.X -
        vQuat.Y * vConj.Y - vQuat.Z * vConj.Z;

        // Rajoita näkyvyyttä maan ja taivaan välille
        if (qNewView.Y > -0.49f && qNewView.Y < 0.49f)
        {
            m_vView.X = m_vPos.X + qNewView.X;
            m_vView.Y = m_vPos.Y + qNewView.Y;
            m_vView.Z = m_vPos.Z + qNewView.Z;
        }
    }

    /// <summary>
    /// Muokkaa näkyvyyttä syötelaitteen arvojen perusteella
    /// </summary>
    public void changeView(float fXcontrol, float fYControl)
    {
        float fYRotation    = 0.0f;
        float fXRotation    = 0.0f;

        Vector3 vLook, vLookNorm, vLookNormUnit;
        vLook               = m_vView - m_vPos;

        //Muuta ainoastaan tarvittaessa
        if ((fXcontrol == 0) && (fYControl == 0))
        {
            return;
        }

        // pyöritä y-akselin ympäri sidottuna peliaikaan
        fYRotation = (float)(fXcontrol) * (mfTimeLapse / 2000.0f);

        fXRotation      = (float)(fYControl) / 50.0f;

        vLookNorm       = Vector3.Cross(vLook, m_vUp);
        vLookNormUnit   = Vector3.Normalize(vLookNorm);
        update_camera_view(fXRotation, vLookNormUnit);

        Vector3 vYrotation = new Vector3(0.0f, 1.0f, 0.0f);
        update_camera_view(-fYRotation, vYrotation);
    }

    /// <summary>
```

```
/// Päivittää kameran positiota liikkumisen jälkeen
/// </summary>
public void update_cam_pos_and_view(Vector3 vCam, float
f_speed)
{
    f_speed     *= (float)mfTimeLapse;
    vCam        *= f_speed;
    m_vPos.X    += vCam.X;
    m_vPos.Z    += vCam.Z;
    m_vView.X   += vCam.X;
    m_vView.Z   += vCam.Z;
}

/// <summary>
/// Laskee frameiden välisen aikavälin
/// </summary>
public void set_frame_interval(GameTime gameTime)
{
    //Laskee frameiden välisen aikavälin, parantaakseen
    //animointia

    mfTimeLapse = (float)gameTime.ElapsedGameTime.Milliseconds;
}

/// <summary>
/// Liikuttaa kameran positiota ja näkökenttää sivuttain
/// </summary>
public void strafe(float f_cam_speed)
{
    Vector3 vCam = new Vector3(0.0f, 0.0f, 0.0f);
    Vector3 v_unitCam;
    const float kSpeedScale = 0.005f;
    f_cam_speed *= kSpeedScale;

    vCam.Y       = m_vView.Y;
    vCam.X       = m_vView.X - m_vPos.X;
    vCam.Z       = m_vView.Z - m_vPos.Z;

    //hae uusi kamera vektori ja normalisoi se
    v_unitCam = Vector3.Normalize(vCam);
    Vector3 v3Strafe = Vector3.Cross(v_unitCam, m_vUp);

    update_cam_pos_and_view(v3Strafe, f_cam_speed);
}

/// <summary>
/// Liikuttaa kameraa eteen ja taakse
/// </summary>
public void move(float f_cam_speed)
{
    Vector3 vCam = new Vector3(0.0f, 0.0f, 0.0f);
    Vector3 v_unitCam;
    const float kSpeedScale = 0.005f;
    f_cam_speed *= kSpeedScale;

    vCam.Y = m_vView.Y;
    vCam.X = m_vView.X - m_vPos.X;
    vCam.Z = m_vView.Z - m_vPos.Z;

    //hae uusi kamera vektori ja normalisoi se
```

```
            v_unitCam = Vector3.Normalize(vCam);

            update_cam_pos_and_view(v_unitCam, f_cam_speed);
        }
    }
}



//--------------------------------------------------------
// Esimerkki Peliprojekti - BasicShader.fx tiedoston sisältö
//--------------------------------------------------------
float4x4 fx_WVP : WORLDVIEWPROJ;

struct gtVertex
{
    float4 f4Position : POSITION0;
    float4 f4Color : COLOR0;
};

struct gtPixelFormat
{
    float4 f4Position : POSITION0;
    float4 f4Color : COLOR0;
};

struct tScreenOutput
{
    float4 f4Color : COLOR0;
};

// muokkaa verteksi syötetietoja
void vertex_shader(in gtVertex IN, out gtPixelFormat OUT)
{
    // muunna verteksiä
    OUT.f4Position = mul(IN.f4Position, fx_WVP);
    OUT.f4Color = IN.f4Color;
}
// muokkaa vs output paluutietoja ja lähetä pikseli kerrallaan
//laitteistolle
void pixel_shader(in gtPixelFormat IN, out tScreenOutput OUT)
{
    float4 fColor = IN.f4Color;
    OUT.f4Color = clamp(fColor, 0, 1);
}

technique simple
{
    pass p0
    {
        //Määrittele ja alusta verteksivarjostin
        vertexshader = compile vs_1_1 vertex_shader();
        //Määrittele ja alusta pikselivarjostin
        pixelshader = compile ps_1_1 pixel_shader();
    }
}
```

```
//------------------------------------------------------------
// Esimerkki Peliprojekti - IntroShader.fx tiedoston sisältö
//------------------------------------------------------------
float4x4 fx_WVP : WORLDVIEWPROJ;
float    fx_Blue;
float  fx_PosX;

// verteksivarjostin syötetiedot
struct VS_INPUT
{
      // verteksitiedot XNA koodista
      float4 f4Position : POSITION0;
      float4 f4Color : COLOR0;
};

struct VS_OUTPUT
{
      //paluutiedot verteksivarjostimelta pikselivarjostimelle

      // siirtyy myös Graphics pipelinelle
      float4 f4Position : POSITION0;

      float4 f4Color : COLOR0; //muokataan pikselivarjostimella
};

struct PS_OUTPUT
{
      // Paluutietona lähtee väritetty pikseli
      float4 f4Color : COLOR0;
};

float4 changeBlueValue()
{
      //Muuta väriä
      float4 f4col;
      f4col.r=0.0f; f4col.g=0.0f; f4col.b=fx_Blue; f4col.a=1.0f;
      return f4col;
}

float4 changePosition(float4 f4Position)
{
      //Muuta sijaintia
      float4 f4Pos       = f4Position;
      f4Pos.x                 += fx_PosX;
      return                     f4Pos;
}

// muokkaa verteksivarjostimen syötetietoja
void vertex_shader(in VS_INPUT IN, out VS_OUTPUT OUT)
{
      float4 f4Pos    = changePosition(IN.f4Position);
      OUT.f4Position    = mul(f4Pos, fx_WVP);
      OUT.f4Color       = IN.f4Color * changeBlueValue();
}

// muokkaa verteksivarjostimen paluutietoja ja lähetä pikseli kerral-
//laan laitteistolle
void pixel_shader(in VS_OUTPUT IN, out PS_OUTPUT OUT)
```

```
{
      OUT.f4Color = IN.f4Color;
}

technique simple
{
      pass p0
      {
            vertexshader = compile vs_1_1 vertex_shader();
            pixelshader = compile ps_1_1 pixel_shader();
      }
}


//-----------------------------------------------------------
// Esimerkki Peliprojekti - TextureShader.fx tiedoston sisältö
//-----------------------------------------------------------
float4x4 fx_WVP : WORLDVIEWPROJ; // WVP matriisi
uniform extern texture fx_Texture;      // Muuttuja tekstuurille

sampler textureSampler = sampler_state
{
    Texture = <fx_Texture>;
    magfilter = LINEAR; // magfilter when bigger than actual size
    minfilter = LINEAR; // minfilter when smaller than actual size
    mipfilter = LINEAR;
};

// verteksivarjostimen syötetiedot
struct VS_INPUT
{
    float4 f4Position                       : POSITION0;
    float4 f4Color                          : COLOR0;

    // säilyttää uv-koordinaatteja
    float2 textureCoordinate              : TEXCOORD0;
};

// verteksivarjostimen paluutiedot
struct VS_OUTPUT
{
    float4 f4Position                       : POSITION0;
    float4 f4Color                          : COLOR;
    float2 textureCoordinate                    : TEXCOORD0;
};

// pikselivarjostimen paluutiedot
struct PS_OUTPUT
{
    float4 f4Color : COLOR0;
};

// muokkaa verteksivarjostimen syötetietoja
void vertex_shader(in VS_INPUT IN, out VS_OUTPUT OUT)
{
    OUT.f4Position = mul(IN.f4Position, fx_WVP);
    OUT.f4Color = IN.f4Color;
    OUT.textureCoordinate = IN.textureCoordinate;
}
```

```
// muokkaa verteksivarjostimen paluutietoja ja lähetä pikseli kerral-
//laan laitteistolle
void pixel_shader(in VS_OUTPUT IN, out PS_OUTPUT OUT)
{
    OUT.f4Color = tex2D(textureSampler, IN.textureCoordinate);
    OUT.f4Color *= IN.f4Color;
}

technique mytechnique
{
    pass p0
    {
        // alustetaan tekstuurisampleri
        sampler[0] = (textureSampler);

        // declare and initialize vs
        vertexshader = compile vs_1_1 vertex_shader();

        // declare and initialize ps
        pixelshader = compile ps_1_1 pixel_shader();
    }
}
```