

**Virtuaalimaailmojen toteuttamiseen käytettävät vaihtoehdot:
tarkastelussa VRML, OpenGL ja Java 3D**

Toni Kovaniemi

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Pro gradu -tutkielma
Kesäkuu 2002

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

Toni Kovaniemi: Virtuaalimaailmojen toteuttamiseen käytettävät vaihtoehdot:

tarkastelussa VRML, OpenGL ja Java 3D

Pro gradu -tutkielma, 72 sivua, 12 liitesivua

Kesäkuu 2002

Tässä tutkielmassa käsitellään virtuaalimaailmojen toteuttamiseen soveltuvia vaihtoehtoja. Tarkastelussa ovat VRML, OpenGL ja Java 3D. Lisäksi tutkielmassa käydään läpi lyhyesti myös muita vaihtoehtoja. Tutkielmassa esitellään jokaisella vaihtoehdolla toteutettuja sovelluksia. Tutkielmaa varten on jokaisella kolmella vaihtoehdolla toteutettu myös esimerkkisovelluksia, joiden toteuttamisprosesseja käsitellään tutkielmassa. Esimerkkisovelluksilla mitattiin myös OpenGL- ja Java 3D -rajapintojen välisiä piirtonopeuseroja. Lisäksi tutkielmassa käsitellään myös kolmiulotteisen tietokonegrafiikan piirtämiseen liittyvää termistöä ja eri tekniikoita.

VRML-kuvauskielellä on mahdollista toteuttaa virtuaalimaailmoja nopeasti ilman ohjelmointitaitoa, kun taas Java 3D ja OpenGL vaativat aikaisempaa ohjelmointikokemusta ja tuntemusta tietokonegrafiikasta. Interaktiivisten mallien toteuttamiseen VRML ja Java 3D soveltuvat OpenGL-rajapintaa paremmin, sillä siinä ei ole valmiita ominaisuuksia interaktiivisten mallien toteuttamiseen. OpenGL soveltuu parhaiten graafisesti raskaiden ja nopeutta vaativien ympäristöjen piirtämiseen.

Avainsanat ja -sanonnat: VRML, OpenGL, Java 3D, virtuaalimaailma, tietokonegrafiikka, piirtonopeus.

Kiitokset

Kiitos Martti Juholalle graduni asiantuntevasta ja kannustavasta ohjauksesta sekä Timo Tossavaiselle asiantuntevista neuvoista. Kiitos vanhemmille ja sis-kolle tuesta sekä kannustuksesta. Suuri kiitos myös ystäville, jotka antoivat mi-nulle välillä myös muutakin ajateltavaa.

Erityiskiitos Nooralle kadonneiden pilkkujen ja kirjoitusvirheiden metsäs-tyksestä.

Sisällys

1.	Johdanto.....	1
2.	Virtuaalimaailmojen historiaa.....	3
2.1.	Sensorama.....	3
2.2.	SIMNET.....	3
2.3.	Lucasfilm's Habitat.....	4
2.4.	AlphaWorld.....	5
2.5.	Tietokonepelit.....	6
3.	Kolmiulotteisen grafiikan peruskäsitteistöä ja -tekniikoita.....	7
3.1.	Geometriset peruselementit.....	7
3.2.	Koordinaatisto ja objektien affiinimuunnokset.....	7
3.2.1.	Kartesinen koordinaatisto.....	7
3.2.2.	Affiinimuunnokset.....	8
3.3.	Yksinkertaisia valaistusmalleja.....	9
3.3.1.	Malleissa käytettäviä valonlähteitä.....	9
3.3.2.	Diffuusi ja spekulari heijastus.....	10
3.4.	Piirtotekniikat ja realismi.....	11
3.4.1.	Tasasävytyt.....	11
3.4.2.	Gouraud-sävytyt.....	11
3.4.3.	Phong-sävytyt.....	12
3.4.4.	Radiositeetti.....	12
3.4.5.	Teksturointi.....	13
3.4.6.	Pinnan kuhmuttaminen.....	13
3.4.7.	Reunanpehmennys.....	14
4.	Käsiteltävien kielten ja rajapintojen perusominaisuudet.....	16
4.1.	Johdanto.....	16
4.2.	VRML.....	17
4.2.1.	Historia.....	17
4.2.2.	Toimintaperiaate.....	18
4.2.3.	Solmut ja objektit.....	19
4.2.4.	Prototyypit.....	20
4.2.5.	Affiinimuunnokset.....	21
4.2.6.	Ryhmäsolmut ja monistaminen.....	21
4.2.7.	Värit ja tekstuurit.....	22
4.2.8.	Animaatio ja sensorit.....	23
4.2.9.	Kuvakulmat ja valot.....	24
4.2.10.	Linkitys ja ääni.....	25
4.2.11.	Java ja JavaScript.....	25

4.3.	OpenGL.....	26
4.3.1.	Historia.....	26
4.3.2.	Toimintaperiaate.....	26
4.3.3.	Objektin rakentaminen	27
4.3.4.	Matriisit ja affiinimuunnokset	30
4.3.5.	Värit, varjostaminen ja tekstuurit.....	30
4.3.6.	Kaksoispuskurointi	31
4.3.7.	Mallin katseleminen ja perspektiivi.....	32
4.3.8.	Valot ja pintamateriaalit	33
4.3.9.	OpenGL-apukirjastot	34
4.4.	Java 3D API.....	36
4.4.1.	Historia.....	36
4.4.2.	Toimintaperiaate	36
4.4.3.	Maisemagraafi.....	37
4.4.4.	Objektin rakentaminen	38
4.4.5.	Ryhmäsolmut	39
4.4.6.	Värit, varjostusmallit ja tekstuurit	39
4.4.7.	Animaatio ja vuorovaikutus	40
4.4.8.	Mallin katseleminen	41
4.4.9.	Valot.....	42
4.4.10.	Äänet	42
4.4.11.	Syöttölaitteet.....	43
4.5.	Muita soveltuvia kieliä ja rajapintoja.....	43
5.	Ominaisuuksien keskinäinen vertailu	46
5.1.	Käyttöönotto ja käyttäminen	46
5.2.	Mallin rakentaminen ja sen piirtäminen	47
5.3.	Valot ja äänet	48
5.4.	Animaatio ja vuorovaikutus	48
5.5.	Syöttölaitteet.....	49
5.6.	Yhteenveto	49
6.	Esimerkkejä toteutetuista sovelluksista.....	52
6.1.	VRML	52
6.1.1.	Virtuaali-Helsinki	52
6.1.2.	Lääketeolliset sovellukset.....	52
6.2.	OpenGL.....	53
6.2.1.	Suunnitteluohjelmat	53
6.2.2.	Tietokonepelit	54

6.3.	Java 3D.....	55
6.3.1.	Kolmiulotteisten mallien katseluohjelmat	55
6.3.2.	COVET ja COSMOS2	56
7.	Esimerkkisovellusten toteuttaminen.....	58
7.1.	Toteutetut sovellukset.....	58
7.2.	Toteutusprosessit.....	59
7.2.1.	VRML	59
7.2.2.	OpenGL.....	60
7.2.3.	Java 3D	62
7.3.	Piirtonopeuden testaaminen	63
7.3.1.	Johdanto	63
7.3.2.	Testin toteutus.....	64
7.3.3.	Rajapintojen välinen nopeusero	64
7.3.4.	Monikulmioiden lisäämisen vaikutus piirtonopeuteen	66
8.	Lopuksi	67
8.1.	Tulevaisuudennäkymät	67
8.2.	Loppusanat	68
	Viiteluettelo	69
	Liitteet	

Termistö

Abstrakti luokka: Luokka, jolla on avoimia virtuaalioperaatioita. [Koskimies, 2000]

AIF: Audio Interchange File Format. AIF-tiedostomuoto on alun perin Apple-tietokoneiden äänistandardi. [Äänipää, 2001]

Aliluokka: Luokka, joka on perinyt toisen luokan piirteet. [Koskimies, 2000]

API: Application Programming Interface, sovellusliittymä. [ATK-Sanakirja, 1999]

AU: Audio. AU-tiedostomuoto on alunperin Sun-tietokonevalmistajan standardi. [Äänipää, 2001]

Avatar: Pelaajan roolihahmo virtuaalimaailmassa. [ATK-Sanakirja, 1999]

BMP: Microsoft Windows Device Independent Bitmap. Microsoftin oma yksinkertainen kuvaformaatti, joka pakkamaton tai heikolla häviöttömällä pakkausalgoritmeilla pakattu. [Kokkarinen *et al.*, 2001]

FPS: Frames per second. Ilmoittaa kuinka monta ruutua piirretään kuvaruudulle sekunnissa.

GIF: Graphics Interchange Format. Häviötöntä pakkausmenetelmää käyttävä kuvaformaatti, joka tukee enimmillään 256-väristä kuvainformaatiota. [Kokkarinen *et al.*, 2001]

Immersio: eläytyminen, uppoaminen virtuaalimaailmaan

JNI: Java Native Interface -rajapinta, joka mahdollistaa muilla kielillä tehtyjen komponenttien kutsumisen Javasta ja päinvastoin.

JPEG, JPG: Joint Picture Experts Group. Suosittu häviöllistä pakkausta käyttävä kuvaformaatti. [Kokkarinen *et al.*, 2001]

Konveksi monikulmio: Monikulmio, jonka kaksi pistettä voidaan yhdistää monikulmion reunaan leikkaamattomalla janalla.

Liukuväri (engl. gradient): Värin portaaton siirtyminen toiseen väriin kuva-alueen sisällä.

Malli: Useammasta objektista rakennettu piirros. Malli voi olla esimerkiksi talo (seinät ja ikkunat).

MIDI: Musical Instrument Digital Interface. Digitaalinen ohjausliitäntä syntetisaattoreiden ja vastaavien laitteiden välille. Standardi jonka pitäisi varmistaa MIDI-laitteiden yhteensopivuus eri laitevalmistajien välillä.

MPEG: Motion Picture Experts Group. Sarja standardeja digitaalisen audio- ja videomateriaalin tallentamiseen sekä lähettämiseen digitaalisessa muodossa.

MUD: Multi-User Dungeon. Tekstipohjainen roolipeliympäristö usealle pelaajalle. [ATK-Sanakirja, 1999]

Näkymä: Lopullinen näkyvä piirros näytöllä. Näkymä voi koostua useammasta mallista ja objektista, mutta yhtä hyvin näkymässä voi olla vain yksi objekti.

Objekti: Useammasta pisteestä, viivasta, kaaresta tai näiden yhdistelmästä rakennettu kappale. Objekti voi olla esimerkiksi pallo, neliö tai seinä.

Olio: Luokasta luotu ilmentymä. [ATK-Sanakirja, 1999]

Periytyminen: Luokan piirteiden siirtyminen jollekin toiselle luokalle. [Koskimies, 2000]

Pikseli: Digitaalisen kuvan pienin alkio. [ATK-Sanakirja, 1999]

PNG: Portable Network Graphics. Häviötöntä pakkausta käyttävä kuvaformaatti. [Kokkarinen *et al.*, 2001]

Rautalankamalli: Objekti, josta esitetään ainoastaan reunaviivat.

RGB: Tietokoneiden yhteydessä yleisimmin mainittu värijärjestelmä, jossa väri ilmoitetaan punaisen, vihreän ja sinisen komponentin yhdisteenä. [Kurahila, 1997]

RGBA: RGB-värijärjestelmä, johon on lisätty tieto värin läpikuultavuudesta.

SDK: System Development Kit, kehityspaketti. [ATK-Sanakirja, 1999]

Sovelma (engl. applet): Verkkosivuilla oleva Java-ohjelmointikielellä toteutettu ohjelma, joka suoritetaan verkkoselaimessa.

Virkistystaajuus: Ilmoittaa kuinka monta kertaa kuvaa päivitetään sekunnissa.

Välittömän piirtämisen rajapinta: Rajapinta, jossa mallia piirretään sitä mukaa kun rajapinnalle syötetään objektien kuvauksia. [Kokkarinen *et al.*, 2001]

WAV: Waveform Audio File Format. WAV-tiedostomuoto on Microsoftin ja IBM:n Windows 3.1 -käyttöjärjestelmään kehittelemä ääniformaatti. [Äänipää, 2001]

Yliluokka: Luokka, josta aliluokan ominaisuudet peritään. [Koskimies, 2000]

1. Johdanto

Elämme kolmiulotteisessa maailmassa. Luontaista kolmiulotteista havainnointiamme ei kuitenkaan ole hyödynnetty nykyisissä käyttöliittymissä, koska aiemmat laitteistot eivät ole siihen kyenneet. Nyt kuitenkin tämä on mahdollista. Virtuaalimaailmoissa ihmisen toimintaa eivät ole rajoittamassa reaali-maailmasta tutut fysiikan lait. Voimme tehdä siellä asioita, joita emme voisi tehdä todellisuudessa tai niiden toteuttaminen olisi työlästä. Meidän on mahdollista vieraila kaukaisilla planeetoilla tai itse toteuttaa todellisuudesta kokonaan riippumattomia maailmoja. Eri asioiden kolmiulotteisesta esittämisestä on usein myös taloudellista hyötyä. Suunnittelija voi toteuttaa uudesta tuotteesta kolmiulotteisen mallin, jota asiakkaan on mahdollista tarkastella ennen varsinaisen tuotteen valmistamista. Työntekijöitä on mahdollista kouluttaa virtuaalimaailmojen avulla uusiin työtehtäviin ennen kuin työtilat ovat edes valmistuneet.

Kolmiulotteiset maailmat ovat tulleet viime aikoina ihmisille tutuiksi. Tietokoneiden huima kehitys ja hintojen laskeminen on tuonut virtuaalimaailmat myös kotitietokoneisiin. Ne ovat yleistyneet sekä tietokonepeleissä että Internetissä. Lisäksi maailmojen tarjoamia mahdollisuuksia hyödynnetään yhä kasvavassa määrin muun muassa tutkimustyössä ja suunnittelussa. Vanhoilla tietokoneilla yksittäisen kuvan piirtäminen saattoi kestää minuutteja tai jopa tunteja, mutta nykyiset laitteistot pystyvät piirtämään useita kymmeniä kuvia sekunnissa.

Virtuaalimaailmojen toteuttaminen vaatii ohjelmointikieleltä tai -rajapinnalta useita ominaisuuksia. Sen avulla täytyy pystyä toteuttamaan ainakin monipuolista kolmiulotteista grafiikkaa. Usein virtuaalimaailman halutaan sisältävän paljon muutakin kuin kolmiulotteisen näkymän maailmaan. Monet virtuaalimaailmat sisältävät ääntä, ja niissä voi käyttää erilaisia syöttölaitteita kuten esimerkiksi datahansikasta. Käytettävät lisäominaisuudet asettavat myös omat vaatimuksensa virtuaalimaailman toteuttamiseen soveltuvaa vaihtoehtoa valittaessa. Usein saatetaan myös vaatia ohjelmointikieleltä tai -rajapinnalta laiteriippumattomuutta, jolloin virtuaalimaailman siirtäminen toiseen laiteympäristöön helpottuu. Kaikkia vaatimuksia sisältävää, ainoa oikeata vaihtoehtoa ei kuitenkaan ole olemassa. Tällöin joudutaan tapauskohtaisesti kartoittamaan ja valitsemaan ominaisuuksiltaan soveltuvin vaihtoehto, joka sopii kyseisen virtuaalimaailman toteuttamiseen. Mikä eri vaihtoehtoista on minun projektiini sopiva? Minkälaisia ominaisuuksia se tarjoaa käytettäväksi? Minkälaisia virtuaalimaailmoja sillä on jo aikaisemmin toteutettu?

Tässä tutkielmassa käsittelen virtuaalimaailmojen toteuttamiseen soveltuvia vaihtoehtoja. Tarkastelussa ovat VRML, OpenGL ja Java 3D, joiden eri ominaisuuksiin paneudutaan tutkielmassa tarkemmin. Lisäksi tutkielmassa käydään läpi lyhyesti myös muita vaihtoehtoja. Tutkielma tarjoaa tarkasteltavista vaihtoehtoista ja niiden sisältämistä ominaisuuksista kattavan tietopaketin, jota voidaan käyttää apuna valittaessa virtuaalimaailman toteuttamiseen soveltuvaa ympäristöä. Jokaisella vaihtoehdolla olen toteuttanut tutkielmaa varten myös esimerkkisovelluksia, joiden toteuttamisprosesseja käsittelen tutkielmassa. Esimerkkisovelluksilla mitattiin myös OpenGL- ja Java 3D -rajapintojen välisiä piirtonopeuseroja. Lisäksi tutkielmassa käsitellään myös kolmiulotteisen tietokonegrafiikan piirtämiseen liittyvää termistöä ja eri tekniikoita. Tutkielmaan on pääasiassa koottu tietoa ohjelmointikielten ja -rajapintojen määrittelydokumentaatioista. Lisäksi apuna on käytetty tietokonegrafiikkaa sekä virtuaalitodellisuutta ja -ympäristöjä käsittelevää kirjallisuutta.

Tutkielman luvuissa kaksi ja kolme tarkastellaan virtuaalimaailmojen historiaa sekä kolmiulotteisen grafiikan peruskäsitteistöä ja -tekniikoita. Neljännessä luvussa tutustutaan VRML-kuvauskielen sekä OpenGL- ja Java 3D -rajapintojen tarjoamiin ominaisuuksiin sekä esitellään myös muita virtuaalimaailmojen toteuttamiseen soveltuvia vaihtoehtoja. Esiteltyjä ominaisuuksia vertaillaan keskenään tutkielman viidennessä luvussa. Kuudennessa luvussa esitellään eri vaihtoehtoilla toteutettuja sovelluksia. Seitsemännessä luvussa toteutetaan esimerkkisovellukset ja tehdään testi, jolla mitataan OpenGL- ja Java 3D -rajapintojen välistä nopeuseroa esimerkkisovelluksia piirrettäessä. Viimeisessä eli kahdeksannessa luvussa pohditaan myös hieman tulevaisuuden näkymiä.

2. Virtuaalimaailmojen historiaa

Virtuaalimaailmojen historia juontaa juurensa yllättävänkin kauas 1900-luvulle. Ensimmäiset todellisuutta mallintavat simulaatiot tehtiin jo ennen tietokoneiden aikakauden alkamista. Tässä luvussa esitellään virtuaalimaailmojen historian joitakin tärkeimpiä askeleita, aina vuodesta 1956 tähän päivään asti. Virtuaalimaailmoihin liittyvien lisälaitteiden, kuten esimerkiksi pääripusteisten näyttöjen ja datahansikkaiden historia sivutetaan. Niiden historiasta kiinnostuneille suositon esimerkiksi kirjaa “The Science of Virtual Reality and Virtual Environments” [Kalawsky, 1993].

2.1. Sensorama

Vuonna 1956 Morton Heilig kehitti laitteen nimeltä Sensorama (ks. kuva 2.1), joka tarjosi käyttäjälleen mielikuvituksellisen moottoripyörällä ajon Manhattanilla. Sensoraman käyttäjä näki Manhattanista kolmiulotteisen näkymän ja todentuntuisuutta lisättiin tärisevillä ohjaustangon kahvoilla sekä pyörän istuimella. Ajamisesta syntyvää ilmavirtauksen voimakkuutta säädeltiin pyörän ajonopeuden mukaan. Immersion tunnetta lisättiin autojen pakokaasujen hajulla sekä ohitetuista ruokapaikoista leijailevilla tuoksuilla. [Kalawsky, 1993] Heiligin keksintö oli passiivinen simulaattori, josta vähitellen siirryttiin interaktiivisiin simulaattoreihin tietokoneiden kehittyessä.



Kuva 2.1. Sensorama [ArtMuseum.net, 2000]

2.2. SIMNET

Simulaattoreiden ja virtuaalimaailmojen mahdollisuudet huomattiin myös sotilaskäytössä. Ensimmäinen laajamittaisesti virtuaalimaailmaa hyödyntänyt toteutus oli vuosina 1983-1989 toiminut SIMNET, joka oli USA:n puolustusministeriön hanke. SIMNET oli alunperin tarkoitettu lähinnä yksittäisen panssarivaunumiehistön harjoitteluun, mutta sitä laajennettiin simuloi-

maan useiden miehistöjen yhtäaikaista toimintaa taistelutilanteessa. Kuva panssarivaunun ulkopuolisesta näkymästä välitettiin miehistölle televisio-ruutujen välityksellä (ks. kuva 2.2). [Background paper, 1995] 1980-luvun loppupuolen teknologian tehostommuuden vuoksi, SIMNET toimi ainoastaan 640x480 pikselin resoluutiolla, ja päivitti näkymää noin 30 fps, kun näkyvissä oli 100 000 monikulmiota. [Sohl]



Kuva 2.2. Vaunun ajaja SIMNET M1 panssarivaunusimulaattorissa
[Background paper, 1995]

2.3. Lucasfilm's Habitat

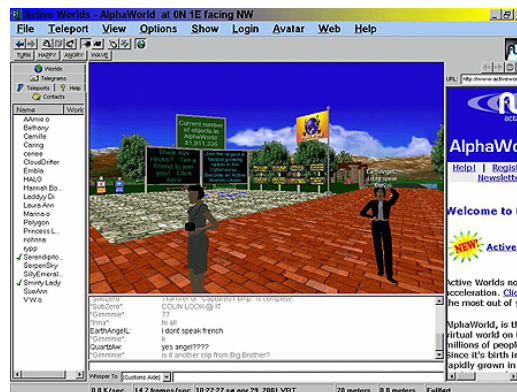
Vuonna 1985 otettiin käyttöön yksi ensimmäisistä usean käyttäjän virtuaali-maailmoista, joka oli avoin kaikille halukkaille. Virtuaalimaailman nimi oli Lucasfilm's Habitat ja se toimi Commodore 64:llä, joka oli tuon ajan yksi suosituimmista kotitietokoneista. Habitatin eri versioita toimi vuosikymmenen loppupuolella kaupallisena palveluna Japanin ja Yhdysvaltojen puhelin-verkoissa, ja käyttäjiä niillä oli Yhdysvalloissa noin 15 000. Toiminnaltaan Habitat oli samantapainen kuin nykyiset monen käyttäjän virtuaali-maailmat; käyttäjällä on avatar, jota pystyy liikuttelemaan ohjaimen avulla virtuaali-maailmassa. Habitatin avatarilla oli mahdollista kerätä maailman objekteja, sekä jutella ja elehtiä toisten avatarien kanssa (ks. kuva 2.3). [Morningstar and Farmer, 1990]



Kuva 2.3. Lucasfilm's Habitat [Morningstar and Farmer, 1990]

2.4. AlphaWorld

Internetin käytön yleistymisen ja sinne rakennettavien kolmiulotteisten maailmojen rakentamisen mahdollistavan VRML-kuvauskielen syntyminen loi Internetiin suuren joukon virtuaalisia maailmoita. Yksi tunnetuimmista ja eniten huomiota saaneista maailmoista on AlphaWorld, joka avattiin yleisölle kesäkuussa 1995 (ks. kuva 2.4). AlphaWorldissä käyttäjillä on mahdollisuus luoda oma avatar, ja elää tämän kautta yhdessä kaikkien muiden AlphaWorldin yhteisöjen avatarien kanssa. Käyttäjä pystyy rakentamaan AlphaWorldin maailmaan oman asuinympäristönsä käyttämällä rakentamiseen suurta joukkoa erilaisia valmisobjekteja aina huonekaluista kasveihin, sekä liikkumaan vapaasti ympäriinsä tällä yli 400 000 neliökilometrin kokoisella alueella. [AlphaWorld, 2001; AlphaWorld Map; Damer, 1996] AlphaWorldissä on ollut vierailijoita ja maailman rakentamiseen osallistuneita käyttäjiä useita tuhansia, ja siellä on järjestetty jopa virtuaalisia häätilaisuuksia, joissa AlphaWorldissä tavanneet ihmiset ovat menneet naimisiin. [Damer, 1996]



Kuva 2.4. AlphaWorld [AlphaWorld, 2001]

2.5. Tietokonepelit

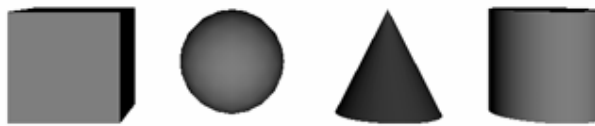
Virtuaalimaailmat ovat nykyisin kaikkien ulottuvilla; tämän on mahdollistanut kotitietokoneiden ja niiden oheislaitteiden huima kehitys. Järvisen [1996] mukaan tavalliselle kotikäyttäjälle tietokonepelit ovat tutuin ovi virtuaalimaailman. Pelejä on monenlaisia; toiset tarjoavat ainoastaan kaksiulotteisen näkyvän maailmaan, kun taas toiset pelit imaisevat pelaajansa mukaansa kolmiulotteisella maailmallaan. Usein pelit tarjoavat pelaajalleen realistisen tunteen nykyaikaisena ralliauton ajamisesta tai tehtävää suorittavasta erikoisjoukkojen sotilaasta. Internetin myötä peleistä on tullut myös moninpelejä, joissa pelaajat voivat olla mistä tahansa päin maapalloa. Pelien virtuaalimaailmat ovat monesti monikansallisia temmelyskenttiä, joissa ei rajoja tunneta. Monissa peleissä immersion tunnetta lisätään pelaajien välisellä kommunikoinnilla, jolloin he voivat yhdessä vaikuttaa pelin kulkuun.

3. Kolmiulotteisen grafiikan peruskäsitteistöä ja -tekniikoita

3.1. Geometriset peruselementit

Nykyisin kolmiulotteisessa grafiikassa kuvataan näkymät lähinnä monikulmioista koostuvien monikulmiomallien avulla. Monikulmiot kuvataan niiden kulmapisteiden avulla. Tällaisia malleja on helppo käsitellä lineaarialgebran keinoin ja piirtoa on helppo nopeuttaa laitteiston avulla. Kaikki monikulmiot voidaan jakaa kolmioihin, joten periaatteessa kolmion nopea piirtäminen eri piirtotekniikoilla riittää.

Monikulmioita yhdistelemällä on mahdollista rakentaa monimutkaisempia kolmiulotteisia kappaleita. 3D-mallinnusohjelmissa ja ohjelmointiympäristöissä tavallisimpia monikulmioista rakennettuja kappaleita ovat muun muassa pallot (engl. sphere), sylinterit (engl. cylinder), kartio (engl. cone) ja laatikko (engl. box/cube) (ks. kuva 3.1).



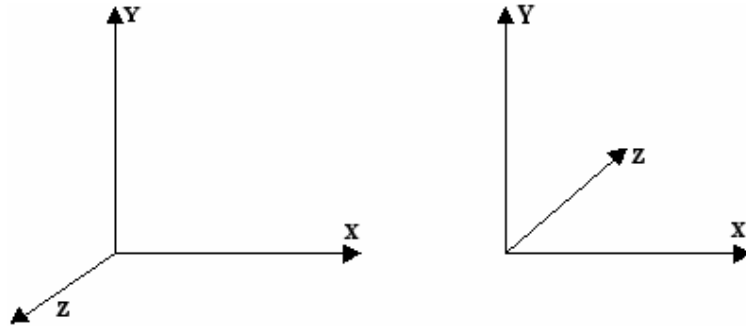
Kuva 3.1. VRML:n peruselementtejä [Carey *et al.*, 1997]

3.2. Koordinaatisto ja objektien affiinimuunnokset

3.2.1. Karteesinen koordinaatisto

Koordinaatistolla on olennainen rooli kolmiulotteisessa mallintamisessa. Sen avulla määritetään yksittäisen objektin sijainti avaruudessa, ja vertaillaan objektien etäisyyksiä ja suhteita toisiinsa. [Hintikka *et al.*, 1998] Karteesinen koordinaatisto koostuu kolmesta akselista, joista jokainen akseli on kahta muuta akselia vastaan kohtisuorassa [Vince, 1995].

Useimmiten 3D-ohjelmissa käytetty koordinaatisto on niin sanottu oikean käden koordinaatisto, jossa x-akseli saa positiiviset arvonsa origon oikealla puolella, y-akseli origon yläpuolella ja z-akseli origon etupuolella [Hintikka *et al.*, 1998]. Vasemman käden koordinaatistossa z-akseli saa positiiviset arvonsa origon takana (ks. kuva 3.2) [Slater *et al.*, 2001].



Kuva 3.2. Oikean käden koordinaatisto (vas.) ja vasemman käden koordinaatisto

3.2.2. Affiinuunnokset

Affiinuunnokset (engl. affine transform) ovat tietokonegrafiikan tuottamisessa erittäin hyödyllinen joukko avaruuden kuvauksia. Affiinuunnokset saadaan suorittamalla peräkkäin jokin mielivaltainen jono perusuunnoksia, jotka voivat olla siirtoja (engl. translation), skaalauksia (engl. scaling), rotaatioita (engl. rotation) ja leikkureita (engl. shear). Ne on mahdollista laskea tehokkaasti, sillä kukin affiinuunnos voidaan kätevästi esittää yhtenä matriisikertolaskuna. Kahden affiinuunnoksen yhdiste on aina affiinuunnos. Kolmiulotteisen avaruuden affiinuunnokset voidaan esittää 4×4 -matriiseina neliulotteisten niin sanottujen homogeenisten koordinaattien avulla.

Jokainen affiinuunnos on bijektio, eli se kuvaa jokaisen avaruuden pisteen tarkalleen yhteen pisteeseen niin, että kaksi alun perin toisistaan eroavaa pistettä kuvautuvat kahdeksi eri pisteeksi. Jokaiselle affiinuunnokselle on tämän vuoksi olemassa sitä vastaava käänteismuunnos, jonka käyttäminen kumoaa alkuperäisen muunnoksen vaikutuksen. Kaikki matriisien esittämät kuvaukset eivät ole bijektioita. [Kokkarinen *et al.*, 2001]

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Kuva 3.3. X-kiertomatriisi (vas.), siirtomatriisi ja skaalausmatriisi [Brits, 2001]

3.3. Yksinkertaisia valaistusmalleja

3.3.1. Malleissa käytettäviä valonlähteitä

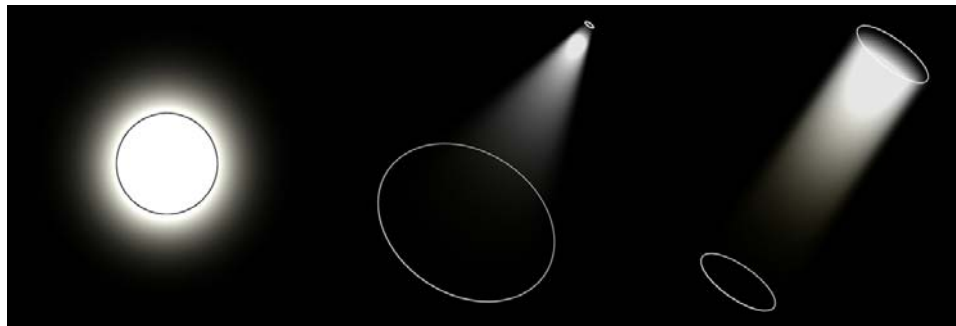
Mallin valaisemisella on tärkeä osa mallin näytävyydessä. Ilman mallissa olevia valonlähteitä upeimmatkin rakennelmat olisivat katsojalle näkymättömiä. Tässä kappaleessa käsitellään kolmiulotteisissa malleissa käytettävien yleisempien yksinkertaisten valaisumallien perusominaisuudet. Virtuaalimaailmojen valaiseminen voidaan jakaa kahteen eri tapaan. Joko käytetään vakiovalaistusta, jolloin kaikkialla mallissa valon voimakkuus on yhtä suuri ja valon tulosuunta aina sama, tai vaihtoehtoisesti lisätään malliin myös muita yksittäisiä valonlähteitä, jotka valaisevat paikallisesti tietyllä alueella.

Pistevalo (engl. point light) on yksinkertainen valonlähde. Siinä valonlähteelle määritellään paikka avaruudessa sekä intensiteetti jolla se valaisee mallia. Pistevalon tehtävä on säteillä valoa yhtä tehokkaasti joka suuntaan kolmiulotteisessa avaruudessa (ks. kuva 3.4). Pistevaloon yleisesti liittyviä ominaisuuksia ovat sen voimakkuus ja väri. Lisäksi usein sille määritellään ominaisuuksia, jotka vaikuttavat valonvoimakkuuden heikkenemiseen pistevalosta etäännyttäessä.

Kohdevalo (engl. spot light) on valonlähde, joka lähettää valokeilan (ks. kuva 3.4). Valokeilalle on määriteltävissä sen suunta koordinaattiakselien suhteen. Se on tyypillisesti kartion muotoinen ja sen huippukulman suuruus on yleensä mahdollista määritellä. Kohdevalon muut ominaisuudet ovat yhteneväiset pistevalon kanssa.

Suuntaavien valonlähteiden (engl. directional light) oletetaan olevan niin kaukana valaistavasta objektista, että kaikki sen lähettämät valon säteet ovat yhdensuuntaisia (ks. kuva 3.4). Aurinko on hyvä esimerkki yhdensuuntaisesta valosta. Muuten suuntaavien valonlähteiden ominaisuuksien voidaan katsoa olevan yhteneväiset pistevalon kanssa.

Hajavallo (engl. ambient light) ei ole suoranainen valo, vaan asetus, jonka avulla määritellään mallin valoisuus. Hajavallo valaisee mallia tasaisesti kauttaaltaan ja se on keskimäärin 20-25 % kokonaisvalaistuksen määrästä. Hajavalolla saadaan mallissa olevien objektien kontrastit ja värit esille. [Vince, 1995] Kokkarisen *et al.* [2001] mukaan hajavalon suurin ongelma on sen tasalautisuus. Objektien valaistujen ja varjossa olevien pintojen väliset sävyerot voivat olla häiritsevän suuria. Yksi vaihtoehto hajavalon käytölle on käyttäjään ”kiinnitettävä” otsalamppu (engl. headlight), joka valaisee kappaleista ne alueet, jotka käyttäjä näkee. Näin mikään näkyvä alue ei ole koskaan käyttäjälle täysin varjossa.



Kuva 3.4. Pistevalo (vas.), kohdevalo ja suunnattu valonlähde [3D Dictionary]

3.3.2. Diffuusi ja spekulaaari heijastus

Mallissa käytetyt valonlähteet vaikuttavat siihen miltä objektit näyttävät katsojalle. Valaistuksen realistista käyttäytymistä simuloivissa reaaliaikaisissa malleissa jouduttaisiin laskemaan lukuisia objektien pinnoista heijastuneita ja taittuneita valonsäteitä ja niiden voimakkuuksia. Tällaisen tietomäärän laskeminen on tietokoneelle erittäin raskasta. Tästä syystä malleissa käytetyn valaistuksen suhteen on tehtävä erittäin suoria yksinkertaistuksia. Reaaliaikaisissa malleissa käytetään yleensä tekniikkaa, jossa huomioidaan ainoastaan valonlähteistä tuleva valo. Muista objekteista heijastuvaa valoa ei huomioida objektin pisteiden valoarvoja laskettaessa. Yksinkertaistamisen takia valaistuksen realismi heikkenee, mutta ilman sitä heijastuksia ei pystyttäisi piirtämään reaaliajassa.

Kaksi yleisesti kolmiulotteisessa grafiikassa käytettyä heijastusta ovat diffuusi ja spekulaaari heijastus. Diffuusi heijastus (Lambertin heijastus) on ehkä yksinkertaisin käytetyistä valaistusmalleista. Siinä objekti heijastaa valoa matkapinnan tavoin siten, että valonlähteestä objektin pisteeseen tuleva valo heijastuu tasaisesti joka suuntaan. Objektin katsomissuunnalla ei ole vaikutusta siihen, kuinka kirkkaasti objektin piste katsojalle näkyy. Pisteeseen kirkkauteen vaikuttaa ainoastaan valonlähteen etäisyys objektista.

Spekulaarilla heijastuksella pyritään mallintamaan valon heijastumista peilimäisestä pinnasta. Siinä objektin valaistava piste heijastaa valoa eniten tulevan valonsäteen suuntaan. Valon heijastuminen muihin suuntiin on sitä heikompa mitä enemmän heijastussuunta poikkeaa valonsäteen suunnasta. Spekulaarisesti heijastavalle objektille tyypillinen ilmiö on pinnalla selkeästi näkyvän heijastuskohta (engl. specular highlight), joka näkyy katsojalle kirkkaampana kuin sitä ympäröivät pisteet. Objektista heijastuvan valon väri on määriteltävissä pinnan ominaisuuksilla, heijastuva valo voi olla pinnan tai valonlähteestä saapuvan valon värinen. Tällä tavalla katsojalle luodaan vaiku-

telma joko muovimaisesta tai metallisesta kappaleesta. [Kokkarinen *et al.*, 2001] Kuvassa 3.5 esitellään diffuusisti ja spekulaaristi heijastavan kappaleen ero.



Kuva 3.5. Diffuusisti ja spekulaaristi heijastavan kappaleen ero [Kokkarinen *et al.*, 2001]

3.4. Piirtotekniikat ja realismi

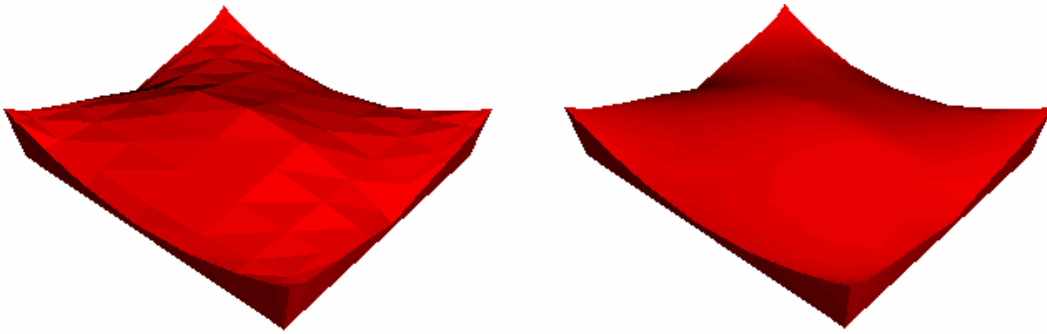
3.4.1. Tasasävytys

Tasasävytyksessä (engl. flat shading) lasketaan valon intensiteetti monikulmion keskellä, ja tätä intensiteettiarvoa käytetään koko monikulmion väriytykseen. Tasainen varjostus on yksinkertainen laskea, mutta sitä käytettäessä monikulmioiden väliset reunat ovat vielä nähtävissä kolmiulotteisissa kappaleissa (ks. kuva 3.6). Tämä voi tosin olla tarkoituksellista piirrettäessä esimerkiksi kulmikkaita kappaleita, kuten esimerkiksi timanttia. Pienempiä monikulmioita käyttämällä, kuvaan saadaan realistisempi vaikutus. Tällöin myös laskenta-aika kasvaa vastaavasti. [Kurhila, 1997; Vince, 1995]

3.4.2. Gouraud-sävytys

Gouraud-sävytys (engl. Gouraud shading) on yleisimmin käytetty monikulmioiden sävytysmalli. Se on menetelmä, jossa valon intensiteettiä monikulmion interpoloidaan eri kohdissa kärkipisteistä laskettujen valoarvojen avulla. Kärkipisteiden välissä olevien reunaviivojen pikselien värit interpoloidaan ensimmäiseksi kulmapisteiden arvoista. Monikulmion sisäpisteiden värit interpoloidaan tämän jälkeen sen pikselirivin reunapikseleistä, jolla laskettava piste sijaitsee. Lineaarinen interpolointi tehdään erikseen jokaiselle värikomponentille (punainen, vihreä ja sininen), ja tämän useampivaiheisen interpoloinnin tuloksena saadaan aikaiseksi liukuvärillä väritetty monikulmio. Tekniikkana Gouraud-sävytys on tehokas ja sen avulla saadaan muun muassa useasta monikulmiosta koostuva pinta vaikuttamaan kaarevalta. Sävytyksen avul-

la pystytään myös luomaan yksinkertaisia syvyysvaikutelmia (ks. kuva 3.6). [Elias, 2001; Kokkarinen *et al.*, 2001; Vince, 1995]



Kuva 3.6. Tasasävytetyistä (vas.) ja Gouraud-sävytetyistä monikulmioista koostuvat pinnat [Elias, 2001]

3.4.3. Phong-sävytys

Phong-sävytyksessä käytetään interpolointia monikulmion normaalivektorin laskemiseen. Menetelmän pääperiaate on laskea monikulmion jokaiselle kärkipisteelle pintaa kohtisuorassa oleva normaalivektori ja interpoloida normaalit monikulmion jokaiselle pikselille. Tämän jälkeen jokaiselle pikselille lasketaan kirkkausarvo kyseisen normaalin perusteella. [Vince, 1995; Elias, 2001]

Tämä menetelmä on laskennallisesti raskas, eikä sitä ei voida tehdä reaaliaikaisesti kotitietokoneilla. Valaistuksiltaan yksinkertaisissakin malleissa Phong-sävytys on huomattavasti Gouraud-sävytystä hitaampi. Kokkarisen *et al.* [2001] mukaan Phong-sävytyksen laskemista on mahdollista kuitenkin nopeuttaa esimerkiksi siten, että pistevektorit lasketaan vain vaakarivin joka toiselle pikselille. Väliin jäävien pikselien värit interpoloidaan niiden vasemmalla ja oikealla puolella olevien pikselien väriarvoista. Eliuksen [2001] mukaan ohjelmat, jotka väittävät käyttävänsä Phong-sävytystä, käyttävät todellisuudessa sävytyksen vastaavia optimoituja sekä kevyempiä muotoja, jotka ovat laskennallisesti huomattavasti nopeampia. Tuloksellisesti ne ovat kuitenkin lähes yhtä hyviä.

3.4.4. Radiositeetti

Radiositeetti (engl. radiosity) on uusin yleisesti käytössä oleva valaistuksen mallintamistekniikka. Se perustuu diffuusien heijastusten käyttöön globaalina valaistusmallina ja simuloi erilaisten pintojen välisiä heijastuksia. Menetelmä perustuu lämpösäteilyn mallintamiseen, ja siinä lasketaan valonlähteistä tulevan energian jakautumista ympäristöön. Radiositeetilaskelmaa tehtäessä mallin kappaleista ja niiden välisistä suhteista muodostetaan lineaarinen yhtälöryhmä, jonka ratkaisemiseksi käytetään yksinkertaista eliminointimenetelmää.

Lopuksi suoritetaan varsinainen kuvan muodostaminen käyttäen esimerkiksi intensiteettiä interpoloivaa valaistusmallia. [Kokkarinen *et al.*, 2001; Kurhila, 1997; Slater *et al.*, 2001]

Kun mallin valaistus- tai heijastusparametreja muutetaan, yhtälöryhmää ei tarvitse muodostaa uudelleen. Mallin katselupistettä muutettaessa vain näkyvän kuvan piirtäminen on tehtävä uudestaan. Vincen [1995] mukaan radiositeettimenetelmän tärkeimpiä ominaisuuksia on juuri sen riippumattomuus katselijasta. Paljon laskentatehoa tietokoneelta vaativat radiositeettilaskelmat on mahdollista tehdä ennen virtuaalimaailman käyttöä, ja saatuja tuloksia voidaan käyttää mallin piirtämisessä.

3.4.5. Teksturointi

Teksturointi (engl. texture mapping) on tekniikka, jolla kolmiulotteinen pinta pinnoitetaan kaksiulotteisella tekstuurilla. Menetelmää voisi hyvinkin verrata tapetin seinälle laittamiseen. Teksturoitavaan monikulmioon liittyy niin sanottu tekstuurikuvaus, joka kuvaa monikulmion jokaisen pisteen kaksiulotteiseksi tekstuurikoordinaateiksi. Käytännössä tekstuurikuvaus määritellään antamalla monikulmion kulmapisteille niitä vastaavat pisteet tekstuurista. Tämän jälkeen lopuille monikulmion pikseleille määritetään niitä vastaavat pisteet lineaarisesti interpoloimalla. Monikulmion jokainen piste saa väriarvonsa tekstuurikuvaukselta, joka määrittää monikulmion pikselille väriarvon sitä vastaavalta tekstuurin alueelta. [Kokkarinen *et al.*, 2001; Vince, 1995]

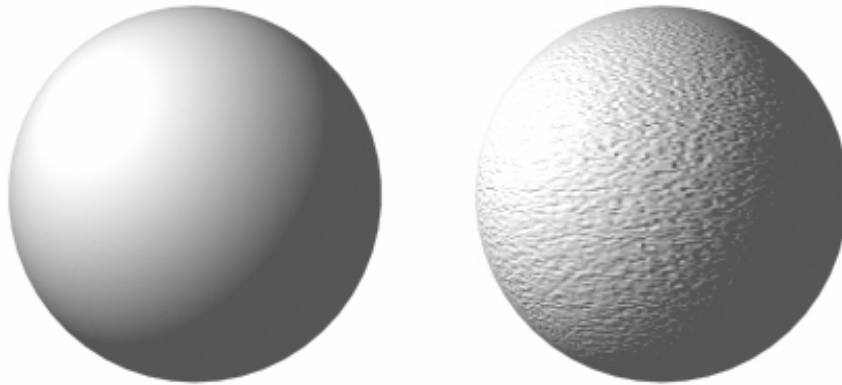
Teksturoinnissa käytettävät tekstuurit sijaitsevat yleensä kuvatiedostoissa, joista ne ladataan käytettäväksi monikulmioiden teksturointiin. Tekstuurien tallentamiseen käytettävät kuvaformaattit voidaan jakaa kahteen ryhmään, häviöttömiin ja häviölliseen. Häviöttömiä kuvaformaatteja ovat muun muassa GIF ja PNG. Näitä kuvaformaatteja käytetään silloin kun kuvainformaation ei haluta muuttuvan. Yleisin käytetty häviöllinen kuvaformaatti on JPG. Sitä käytetään pakkaamaan erityisesti valokuvia ja muita kooltaan suuria kuvia. Häviöllinen kuvaformaatti ei säilytä alkuperäistä kuvainformaatiota sellaisenaan, vaan pakkaukseen käytettävä algoritmi yrittää hävittää sitä mahdollisimman vähän. [Kokkarinen *et al.*, 2001]

3.4.6. Pinnan kuhmuttaminen

Pinnan kuhmuttamisella (engl. bump mapping) on mahdollista luoda realistisen tuntuisia kolmiulotteisia pintoja monikulmioille (ks. kuva 3.7). Menetelmässä monikulmion pinnalle liitetään kuva teksturoinnin tavoin, mutta lisäksi pinnan normaalit lasketaan käyttämällä pinnoitetusta kuvasta saatua tietoa. Kuhmutusefekti ei ole täydellinen, sillä epätasaisuutta ei näy piirretyn kap-

paleen siluettissa eivätkä kuhmut langeta varjoja toisiinsa. [Kokkarinen *et al.*, 2001; Vince, 1995]

Kuhmutuksessa käytettävä tekniikka on samantapainen kun Phong-sävytyksessä, mutta kuhmutuksessa tekstuurin pikselien kirkkausarvot vaikuttavat objektin normaaleihin. Ideaalilanteessa kuhmuttamisalgoritmi yhdistetään pinnan piirtämiseen Phong-sävytyksellä, mutta nykyiset laitteistot eivät kykene reaaliaikaiseen Phong-sävytykseen. Tätä voidaan kuitenkin simuloida kaksiulotteisessa tasossa kohokuvioinnilla, jossa pintaan oletetaan liittyvän korkeuskenttätekstuuri. Se kertoo yksittäisestä pisteestä sen korkeuseron pinnan perustasosta. Korkeuskenttätekstuurin avulla saadaan piirtovaiheessa aikaiseksi vaikutelman kohokuvioidusta kappaleesta. Vaikutelma ei ole kuitenkaan täydellinen, sillä epätasaisuus ei näy esimerkiksi kuhmutetun kappaleen siluettissa eivätkä kuhmut langeta varjoja toisiinsa. [Elias, 2001; Kokkarinen *et al.*, 2001]



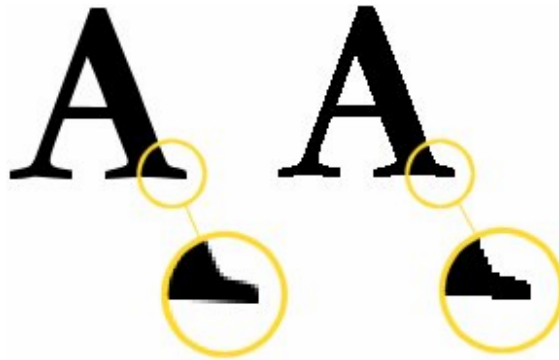
Kuva 3.7. Pinnoittamaton pallo (vas.) ja pinnaltaan kuhmutettu pallo [3D Dictionary]

3.4.7. Reunanpehmennys

Näytölle piirrettävä grafiikka koostuu pikseleistä, joka ovat muodoltaan neli-kulmioita. Tästä pikselin geometrisestä muodosta aiheutuu havaittava grafiikan sahalaitaisuus (ks. kuva 3.8). Ongelma korostuu varsinkin matalilla näyttötarkkuuksilla, joita käytettäessä pikselin koko on suurempi. Suuremmilla näyttötarkkuuksilla ilmiö ei ole niin voimakas, koska pikselit ovat pienempiä. Erityisesti vinoon piirretyt viivat ovat usein porrasmaisia. Kuvanlaadun parantamiseksi sahalaitaisuutta poistetaan käyttämällä niin sanottua reunanpehmennystä (engl. anti-aliasing). Reunanpehmennyksen jälkeen monikulmioiden reunat ovat pehmeämpiä ja hieman utuisempia, mutta huomattavasti paremmalta näyttäviä. [Vince, 1995]

Reunanpehmennys on toteutettavissa usealla eri tavalla. Yksi tapa on laskea vierekkäisten pikseleiden vaikutusta toisiinsa ja pehmittää niiden välisiä värieroja, ja näin lisätä kuvan pehmeyttä. Toinen tapa on käyttää niin sanottu super-sampling -tekniikkaa, jossa kuvaruudulla näkyvä kuva piirretään useita kertoja, kuitenkin jokaisella kerralla hieman eri kohtaan. Tämä pieni kuvan siirtymä pehmentää sahalaitaisuutta.

Reunanpehmennys on laskennallisesti raskas toimenpide, varsinkin jos se tehdään suurissa näytöntarkkuuksissa. Lisäksi muita laskentaan vaikuttavia tekijöitä ovat käytettyjen värien sekä näytöllä olevien monikulmioiden lukumäärä. Reunanpehmennys on mahdollista tehdä ohjelmallisesti, jolloin sen laskemiseen käytetään tietokoneen prosessoriaikaa. Viime aikoina tietokoneissa ovat vakiovarusteiksi yleistyneet 3D-näytönohjaimet, jotka ottavat reunanpehennyksen laskennan oman prosessorinsa tehtäväksi. [Cant *et al.*, 2001; Slater *et al.*, 2001; Vince, 1995]



Kuva 3.8. Pehmennetty (vas.) ja pehmentämätön kirjain [3D Dictionary]

4. Käsiteltävien kielten ja rajapintojen perusominaisuudet

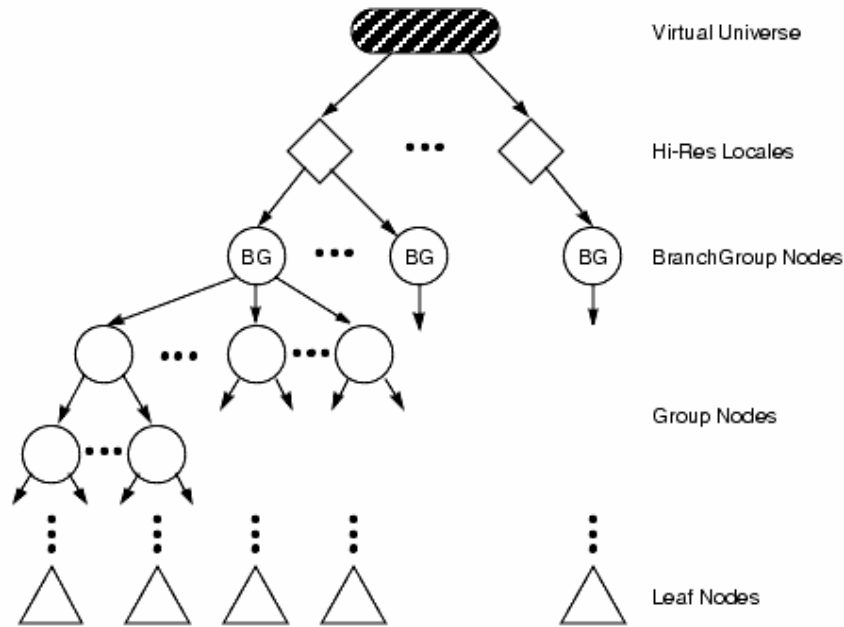
4.1. Johdanto

Tämän luvun tarkoitus on toimia johdatuksena VRML-kuvauskielen, OpenGL- ja Java 3D API -rajapinnan perusteisiin, ja esitellä niitä ominaisuuksia, joita virtuaalimaailman rakentamisessa yleensä tarvitaan. Tarkasteluissa käytetään VRML 2.0:n, OpenGL 1.2:n ja Java 3D API 1.2:n tarjoamia ominaisuuksia, eikä tarkasteluissa oteta huomioon kielten aikaisempien versioiden ominaisuuksia. Myös kielten syntaksin tarkastelu ohitetaan. Kielten syntaksista kiinnostuneille suositan tutustumista kielten määrittelydokumentteihin ja referensseihin [Carey *et al.*, 1997; Woo *et al.*, 1999; Sun OpenGL; Java 3D API, 2000]. Ominaisuuksien esittelyssä keskitytään virtuaalimaailman rakentamisessa tarvittavien keskeisimpien ominaisuuksien tarkasteluun. Näitä ovat muun muassa objektien rakentaminen, affiinimuunnosten tekeminen, värien ja tekstuurien käyttäminen, valot, äänet ja vuorovaikutteisuuden luominen. Lisäksi tarkastellaan eri vaihtoehtojen toimintaperiaatteita, sillä ne eroavat toisistaan huomattavasti.

Ennen ominaisuuksiin perehtymistä on syytä tarkastella tapoja, joilla mallit VRML-kuvauskielessä sekä OpenGL- ja Java 3D -rajapinnoilla piirretään. OpenGL on niin sanottu välittömän piirtämisen rajapinta (engl. immediate mode), jossa mallia piirretään sitä mukaa kuin rajapinnalle syötetään objektien kuvauksia. Kun katsojan silmä liikkuu mallissa ja halutaan piirtää uusi kuva uuden sijainnin mukaan, kaikki piirrettävät kappaleet on syötettävä uudelleen rajapinnalle. VRML ja Java 3D ovat niin sanottuja säilyttävän piirtämisen rajapintoja (engl. retained mode). Ne toimivat korkeammalla abstraktiotasolla siten, että niille annetaan niin sanotun maisemagraafin muodossa kuvaus koko mallista ennen kuin ensimmäistäkään kappaletta aletaan piirtämään. Kun maisemagraafi on kertaalleen rakennettu, sen esittämä maailma voidaan piirtää yhä uudelleen miten monta kertaa tahansa ilman, että maailman kuvausta tarvitsisi syöttää uudelleen piirtävälle rajapinnalle. Piirtämisen välissä on mahdollista tehdä esimerkiksi affiinimuunnoksia maisemagraafissa oleviin solmuihin.

Maisemagraafi on hierarkkinen puurakenne, jonka sisältämä malli piirretään kulkemalla rakenne rekursiivisesti alas lehtisolmuihin, joiden sisältämät objektit annetaan käytettävälle välittömän piirtämisen rajapinnalle piirrettäväksi. Maisemagraafin haaroille on mahdollista myös määritellä niin sanottuja rajauslaatikoita. Jos rajauslaatikko on katsojan näkymän ulkopuolella, voidaan kyseinen maisemagraafin haara jättää piirrosta kokonaan väliin. Tällä tavalla

menettelemällä mallin piirtäminen nopeutuu. VRML:n ja Java 3D:n maisemagraafit ovat keskenään hyvin samanlaisia. Molempien maisemagraafi on rakenteeltaan suunnattu sykliton verkko (engl. directed acyclic graph), jossa vanhemman ja lapsisolmun välinen suhde on yksisuuntainen. Kuvassa 4.1 on esimerkki Java 3D:n maisemagraafista. [Java 3D, 2000; Kokkarinen *et al.*, 2001]



Kuva 4.1. Java 3D:n maisemagraafi [Java 3D, 2000]

4.2. VRML

4.2.1. Historia

VRML-kuvauskielen [VRML, 2002] kehityksen katsotaan alkaneen keväällä 1994, jolloin Mark Pesce esitteli ystävälleen Tony Parisille ideansa kolmiulotteisesta käyttöliittymästä Internetiin. Ystävykset rakensivat Mosaic-selaimessa toimivan 3D-käyttöliittymän nimeltä Labyrinth, joka mahdollisti käyttäjänsä siirtyä edestakaisin 3D-ympäristön ja HTML-dokumenttien välillä.

Pesce kutsuttiin esitelmöimään Genevessä järjestettyyn konferenssiin, jonka osallistujat olivat samaa mieltä siitä, että tarvittaisiin yhteinen kieli kuvaamaan kolmiulotteisia maailmoita. David Ragget keksi termin uudelle kielelle: Virtual Reality Mark-up Language. Myöhemmin nimi muutettiin paremmin soveltuvaksi, ja kielen nimeksi tuli Virtual Reality Modelling Language.

VRML:n toteuttamiseen löydettiin useita varteenotettavia tekniikoita. Keskustelujen jälkeen kieltä kehittävä yhteisö päätyi käyttämään runkona Silicon Graphicsin Open Inventor -tiedostomuotoa. Tiedostomuodossa oli tuki geometrinen 3D-maailmojen, valaistusten, pintamateriaalien ja kolmiulotteisten

käyttöliittymien luomiselle. Open Inventorin mahdollisti ainoastaan staattisten ympäristöjen tekemisen. Näissä ympäristöissä pystyi olemaan vain yksi käyttäjä kerrallaan, eikä käyttäjän ja ympäristön välille voitu rakentaa vuorovaikutusta.

VRML 2.0:n kehittämistä varten perustettiin VAG (VRML Architecture Group), joka koostui useista eksperteistä. VRML kehittyi ryhmän ohjauksessa siten, että VRML 2.0 spesifikaatio julkaistiin vuonna 1996. Kielen uusi versio toi keskeisimpinä parannuksina muun muassa animaation, vuorovaikutteisuuden, ja Java- ja JavaScript-yhteensopivuuden. Syksyllä 1996 aloitettiin prosessi VRML 2.0:n standardoimiseksi kansainvälisesti. Myöhemmin vuonna 1997 siitä tuli kansainvälinen standardi: VRML97. [Smith *et al.*, 1996; VRML, 1998]

4.2.2. Toimintaperiaate

VRML-kuvauskieli ei ole ohjelmointikieli, kuten Java tai C++. Se ei myöskään ole HTML:n tapainen merkintäkieli (engl. markup language). Se on mallinnuskieli (engl. modelling language), jolla voidaan kuvata kolmiulotteisia kappaleita sekä luoda kolmiulotteisia maailmoita. VRML-kuvauskielellä tehdyn mallin toteutus on yksinkertaisesti tekstitiedosto, jota tulkitsemalla saadaan mallin graafinen ulkoasu. VRML-malli on hierarkkinen, puurakenteinen, maisemagraafi, jossa on tiedot kaikista mallissa olevista objekteista, ja niiden ominaisuuksiin vaikuttavista muista mallin ominaisuuksista. Malli piirretään kulkemalla maisemagraafi rekursiivisesti juuresta lehtisolmuihin, joiden sisältämät geometriset peruselementit annetaan piirrettäväksi katseluohjelman käyttämälle grafiikkarajapinnalle.

VRML-malleja voi toteuttaa käytännössä kolmella eri tavalla: kirjoittamalla mallin koodi tekstieditorilla, 3D-animaatio- ja mallinnusohjelmalla tai VRML-mallintajalla. Jokaisella tavalla on omat hyvät ja huonot puolensa; esimerkiksi tekstieditorilla kirjoittaminen on hidasta, ja kirjoittajan on syytä tuntea syntaksin kielioppi. Näin tosin varmistetaan koodin oikeaoppisuus ja toimivuus. 3D-mallinnusohjelmien ja VRML-mallintajien etuna on niiden nopeus; käyttäjä näkee ohjelman graafisesta käyttöliittymästä toteuttamansa mallin reaaliajassa, ja tämä mahdollistaa nopeat reagoinnit mahdollisiin virheisiin muun muassa objektien etäisyyksissä tai valaistuksessa. Haittana näissä ohjelmissa voi olla mallin taustalla olevan koodin epästandardisuus tai virheellisyys. [Hintikka *et al.*, 1998]

VRML-tiedosto perustuu kuvan 4.2 mukaiseen rakenteeseen. Jotta VRML-tiedosto tunnistettaisiin esimerkiksi verkkoselaimessa, on ensimmäisellä rivillä oltava otsikko (engl. header). Otsikko on VRML 2.0:ssa muotoa: **#VRML V2.0 utf8 [kommentti] <rivin vaihto>**. Otsikon jälkeen esitellään mallin tilaominaisuuksia, kuten esimerkiksi taustakuvana käytettävät tekstuurit ja

mallin valaistus. Tilaominaisuuksien jälkeen kooditiedostossa ovat solmut, jotka esittelevät mallin objektien muotoja ja ominaisuuksia. Solmujen jälkeen kooditiedostossa määritellään käytettävät prototyypit. Prototyyppien avulla on mahdollista tehdä uudentyyppejä solmuja, jotka perustuvat kuvauskielessä määriteltyihin solmuihin tai toisiin prototyyppihin. Tiedoston lopussa on määritelty mallissa käytettävät tapahtumien käynnistäjät (sensorit kellot, animaatiopolut), sekä kytkimet, jotka ovat rakenteita eri solmujen, käynnistäjien ja niille luotujen tapahtumien välillä. [Hintikka *et al.*, 1998; Carey *et al.*, 1997]

Otsikko (Header)	
Tilaominaisuudet	
Taustatekstuurit	(Background)
Kuvakulmat	(Viewpoint)
Valaistus	(Light)
Solmut (Nodes)	
Prototyypit (Prototypes)	
Käynnistäjät	
Sensorit	(Sensors)
Kellot	(Animation clocks)
Animaatiot	(Animation paths)
Kytkimet (Routes)	

Kuva 4.2. VRML-tiedoston rakenne [Hintikka *et al.*, 1998; Carey *et al.*, 1997]

4.2.3. Solmut ja objektit

VRML-kuvauskieli perustuu pitkälti solmuihin, ja käytettäviä solmutyyppejä VRML 2.0:ssa on eri käyttötarkoituksia varten useita. Muotosolmut määrittelevät mallin objektien muotoja ja niiden ominaisuuksia, ja osa solmuista toimii mallissa esimerkiksi tapahtumia aistivina sensoreina. [Hintikka *et al.*, 1998] Jokaisella solmulla on seuraavat ominaisuudet: solmutyyppi, määrekentät, mahdolliset tapahtumat, toteutus ja solmulle annettu nimi. Solmutyyppejä ovat esimerkiksi muotosolmut kuten *Box* ja *Text*, ja ajan muutoksin reagoiva solmu *TimeSensor*. Määrekenttiä ovat esimerkiksi solmulla *Box* oleva kenttä *size* ja *Text*-solmun *fontStyle*. Solmulle on mahdollista määritellä joukko tapahtumia, joita se voi lähettää toisille solmuille ja ottaa vastaan toisilta solmuilta. Lisäksi jokainen solmu voidaan nimetä yksilöllisesti. [Carey *et al.*, 1997]

Jokainen solmu aloitetaan tyyppimäärittelyllä, ja erityisesti jokaiselle muotosolmulle on hyvä antaa nimi ennen tyyppimäärittelyä käyttämällä DEF-avainsanaa. Nimen avulla on mahdollista myöhemmin kutsua solmua uudel-

leen käytettäväksi tai muutettavaksi. Solmun määrekentät ovat tyyppimäärittelyn jälkeen alkavan aaltosulkuparin sisällä. Määrekentät muodostuvat syntaksisesti siten, että ensin on kentän nimi ja sen perässä kentän arvo. [Hintikka *et al.*, 1998; Carey *et al.*, 1997] Hintikan *et al.* [1998] mukaan solmujen keskinäisellä järjestyksellä kooditiedostossa ei ole suurta merkitystä. Kuitenkin muotosolmuilla luotavien objektien ominaisuuksia määrittelevät muut solmut on hyvä esitellä kooditiedostossa ennen muotosolmun esittelyä. Näitä ovat esimerkiksi tekstuureihin ja ääniin liittyvät solmutyypit.

VRML 2.0:n toteutetun mallin geometrisen ilmeen luovat pääosin geometriset primitiivit eli perusobjektit sekä vapaamuotoiset objektit. Geometrisistä primitiiveistä käytettävissä ovat *Box* (laatikko), *Cone* (kartio), *Cylinder* (sylinderi) ja *Sphere* (pallo). Lisäksi muita muotosolmuja ovat *ElevationGrid* (pintamuodostelma), *Extrusion* (pyörähdyskappale), *IndexedFaceSet*, *IndexedLineSet* (vapaamuotoiset objektit), *PointSet* (piste) ja *Text* (teksti). [Carey *et al.*, 1997] Esimerkissä 4.1 on määritelty yksinkertainen laatikko *Box*-solmun avulla.

Jo primitiivejä yhdistelemällä on mahdollista luoda monipuolisia objekteja, mutta usein malliin tarvitaan myös vapaamuotoisia objekteja. Vapaamuotoisia objekteja voidaan kuvata *IndexedFaceSet*- ja *IndexedLineSet*-solmuilla. Tällöin objektit luodaan malliin koordinaatistoon määritetyn pisteparven perusteella. Haittana tällä tavalla toteutettujen objektien käytössä on se, että pisteparvet saattavat kasvaa monimutkaista objektia mallinnettaessa suuriksi. Tästä seurauksena on kooditiedoston koon kasvaminen, joka vaikuttaa myös sen siirto- ja lukuaikaan. Kooditiedoston suuresta koosta voi olla mallin käytön kannalta haittaa esimerkiksi silloin, kun kooditiedosto sijaitsee Internetissä tai jos käytettävä siirtonopeus ei ole suuri. [Hintikka *et al.*, 1998; Carey *et al.*, 1997]

```
DEF Laatikko Shape {
  geometry Box {
    size 1.0 1.0 1.0
  }
}
```

Esimerkki 4.1 Laatikon määrittely

4.2.4. Prototyypit

Versiosta 2.0 alkaen VRML-kuvauskielessä on ominaisuus, jolla voi luoda solmuista käytettäväksi uusia prototyyppejä. Tällöin käytetään PROTO-rakennetta, jolla luodaan uusi solmutyyppi, joka on muodostettu jo olemassa olevilla solmutyypeillä ja muilla prototyypeillä. Uusi solmutyyppi perii käytettyjen solmujen ominaisuudet. Jokaisella prototyyppillä on oltava oma yksilöllinen nimi, sillä samannimiset prototyypit aiheuttavat kooditiedostoa ajettaessa

ajonaikaisen virheen. Toisissa VRML-kooditiedostoissa oleviin prototyyppien toteutuksiin voidaan viitata EXTERNPROTO-rakenteella. [Carey *et al.*, 1997]

Prototyypeistä on malleja tehdessä suurta käytännön hyötyä. Niiden tehokkaan käytön avulla kirjoitettavan koodin määrä vähenee, kun uudelleen käytetään jo aikaisemmin kirjoitettua koodia. Uudelleenkäytön ansiosta myös VRML-kooditiedoston koko on pienempi, mikä taas näkyy ajassa mikä kuluu mallia siirrettäessä. Lisäksi prototyypeillä voi luoda helposti objektkirjastoja, joita voi käyttää esimerkiksi juuri EXTERNPROTO-rakenteella.

4.2.5. Affiinimuunnokset

Muotosolmun sijaintiin ja asentoihin mallissa voi vaikuttaa *Transform*-solmulla. VRML-kuvauskielessä etäisyyksiä mitataan metreinä ja kulmia radiaaneina. *Transform*-solmua käytetään objektin siirtämiseen, kääntämiseen ja koon skaalaukseen. Solmun *center*-kentällä on mahdollista määritellä uusi piste, jonka suhteen muotosolmun paikkaa ja asentoa muutetaan. Solmun *rotation*-kentän arvoilla käännetään muotosolmun asentoa yhden tai useamman koordinaattiakselin suhteen, ja *scale*-kentän arvoilla objektin kokoa voidaan skaalata. Objektin paikkaa avaruudessa muutetaan antamalla arvoja solmun *translation*-kentälle. [Carey *et al.*, 1997]

Transform-solmun *children*-kenttä mahdollistaa muun muassa sisäkkäisten muunnosketjujen tekemisen. Tällöin tulee kiinnittää erityisesti huomiota siihen missä järjestyksessä objektien siirto-, kääntö- ja skaalausoperaatiot toteutetaan, ja minkä solmun operaatiot käydään ensimmäisenä läpi. Operaatiot suoritetaan aina alimman tason *Transform*-solmusta ylöspäin. Operaatioiden suoritusrjestys on aina seuraava: ensimmäiseksi center, toiseksi scale, kolmanneksi rotation ja viimeiseksi translation. [Vapourtech, 2001]

4.2.6. Ryhmäsolmut ja monistaminen

VRML 2.0 -kuvauskielessä on *Group*-solmu ryhmäsolmujen tekemistä varten. Ryhmäsolmussa voi olla yksi tai useampia solmuja, jotka sijaitsevat *Group*-solmun *children*-kentässä. Ryhmäsolmun käytännöllisyys korostuu varsinkin rakennettaessa useasta solmusta koostuvaa objektia, jota on tarkoitus käyttää mallissa useampaan kertaan. Solmujen ryhmittäminen helpottaa myös useista solmuista koostuvien objektien animaatioiden toteutusta, sillä ryhmäsolmulle tehtävät toimenpiteet vaikuttavat jokaiseen ryhmäsolmussa olevaan solmuun. [Carey *et al.*, 1997; Hintikka *et al.*, 1998]

Solmujen monistaminen tapahtuu VRML 2.0 -kuvauskielessä käyttämällä DEF- ja USE-avainsanoja. DEF- ja USE-avainsanoilla voi kopioida kaikkia solmuja ja solmutyyppejä. Monistamisen etuna on se, että rakennettaessa objektia, joka koostuu useista samanlaisista solmuista tai solmuryhmistä, voidaan

toistuva osa monistaa ja käyttää sen kopioita. Monistamisella pienennetään kooditiedoston kokoa, ja vältytään saman koodin uudelleenkirjoittamiselta. [Carey *et al.*, 1997; Vapourtech, 2001]

4.2.7. Värit ja tekstuurit

Objektin pintamateriaalin ja -tekstuurin määrittelyä varten VRML 2.0 -kuvauskielessä on käytettävissä neljä solmua: *Material*, *ImageTexture*, *MovieTexture* ja *PixelTexture*. Pintamateriaalin tai -tekstuurin ominaisuudet määritellään *Shape*-solmun *appearance*-kentässä käyttämällä *Appearance*-solmua. [Carey *et al.*, 1997] *Material*-solmulla määritellään objektille muun muassa sen diffuusisti heijastama väri (*diffuseColor*), spekulari heijastuksen väri ja voimakkuus (*specularColor* ja *shininess*) sekä objektin läpinäkyvyyden aste (*transparency*) [Carey *et al.*, 1997]. Värien määrittely perustuu RGB-värikanavan, jonka arvot ilmaistaan kolmella lukuarvolla järjestyksessä punainen, vihreä ja sininen. Kukin kanava saa arvot suljetulta välillä 0-1; arvolla nolla värikanava on kokonaan kiinni ja arvolla yksi se on kokonaan auki. Värikanavien arvoja yhdistelemällä saadaan aikaiseksi erilaisia värejä. [Hintikka *et al.*, 1998]

Objektin pintatekstuuri saadaan määriteltyä käyttämällä joko *ImageTexture*, *MovieTexture* tai *PixelTexture*-solmuja. *ImageTexture*-solmua käytetään tavallisen tekstuurin liittämiseen objektin pinnalle [Carey *et al.*, 1997]. Tekstuurin on oltava joko JPG- tai PNG-tiedostomuotoisissa - osa VRML-maailmojen katseluohjelmista hyväksyy tosin myös GIF-tiedostomuotoisen tekstuurin, vaikka se ei ole VRML97-standardin mukaista [Vapourtech, 2001].

MovieTexture-solmulla objektin pinnan pinnoitteeksi on mahdollista antaa MPEG-tiedosto, joka voi sisältää sekä liikkuvaa kuvaa että ääntä. Solmua voidaan käyttää myös pelkän äänen soittamiseen. *PixelTexture*-solmulla tehdään käsin objektin pinnoitteeksi tekstuuri. Lopputulos on käytännössä samanlainen kuin *ImageTexture*-solmulla objektia teksturoidessa, mutta tiedostosta ladattavan tekstuurin sijaan pinnan teksturointi toteutetaan koodaamalla sen pikselien väriarvot kooditiedostoon. [Carey *et al.*, 1997]

Esimerkissä 4.2 on määritelty laatikolle sen diffuusisti ja spekularisti heijastaman valon väri (punainen ja valkoinen), sekä objektin kirkkausaste ja läpinäkyvyys.

```
DEF Laatikko Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 0 0
      specularColor 1 1 1
      shininess 0.5
      transparency 0.0
    }
  }
  geometry Box {
    size 1.0 1.0 1.0
  }
}
```

Esimerkki 4.2 Pintamateriaalin ominaisuuksien määrittely

4.2.8. Animaatio ja sensorit

VRML 2.0 -kuvauskielen keskeisimpiä ominaisuuksia ovat animaatio ja mallin vuorovaikutteisuus. Näiden ominaisuuksien toteuttaminen jakaantuu kahteen osaan: tapahtumien käynnistäjiin ja tapahtumat toisiinsa kytkevään ROUTE-rakenteeseen. Käynnistäjien tehtäviä on muun muassa laskea animaation ja tapahtumien alkuaika sekä kertoa tapahtumat, joiden mukaan animaatio toteutetaan. ROUTE-rakenne ei ole solmu vaan rakenne solmujen ja tapahtumien välillä. Sen avulla kytketään käynnistäjät ja solmut tietoisiksi toistensa tapahtumista. Rakenteen välityksellä siirretään tieto kytkimeltä toiselle, kun tapahtumalle määritelty ehto on toteutunut. Tämän jälkeen tapahtuman seuraava vaihe voi käynnistyä. [Hintikka *et al.*, 1998; Vapourtech, 2001] ROUTE-rakenteella on mahdollista määritellä esimerkiksi tietyin väliajoin tapahtuva objektin siirtyminen, skaalautuminen tai pintamateriaalin muutos. Lisäksi käyttämällä interpolaattoreita (engl. interpolator) voidaan objekteille luoda suoraviivaista animaatiota, kuten esimerkiksi liikeratapolkuja (*PositionInterpolator*) tai värin vaihtelua (*ColorInterpolator*). [Carey *et al.*, 1997; Hintikka *et al.*, 1998]

Tapahtumien luomiseksi on VRML 2.0 -kuvauskielessä käytettävissä useita sensorisolmuja. Sensorisolmut jaetaan ympäristösensoreihin ja käyttäjäsensoriin. Ympäristösensoreita tarkkailevat mallissa tapahtuvia muutoksia, ja käyttäjäsensorit tarkkailevat käyttäjän ja objektin välistä vuorovaikutusta, kuten esimerkiksi kursorin sijaintia, hiiren nappien käyttöä ja hiiren siirtoliikkeitä. [Hintikka *et al.*, 1998] Ympäristösensoreita ovat törmäykset tunnistava *Collision*, ajankulumista tarkkaileva *TimeSensor*, liikkeen tietyllä alueella tunnistava *ProximitySensor* ja objektin näkymistä tarkkaileva *VisibilitySensor*. Lisäksi käyttäjän ja objektin välistä vuorovaikutusta tarkkailevia sensoreita ovat objektin päällä tapahtuvaa osoittimen liikettä ja napin painalluksia tunnistava

TouchSensor, sekä objektin päällä tehtyjä siirtämisliikkeitä tarkkailevat *SphereSensor*, *CylinderSensor* ja *PlaneSensor*. [Carey et al., 1997; Vapourtech, 2001]

Esimerkissä 4.3 on toteutettu laatikko, joka pyörähtää z-akselinsa ympäri kahdessa sekunnissa. *TimeSensor*-solmu (Kello) mittaa ajan kulumista ja se on kytketty *OrientationInterpolator*-solmuun (LaatikonPyöriminen). Interpolaattorin arvon muuttuessa muutetaan vastaavasti *Transform*-solmun (Objekti) asentoa.

```

DEF Objekti Transform {
  children DEF Laatikko Shape {
    ...
  }
}
DEF Kello TimeSensor {
  loop TRUE
  cycleInterval 2.0
}
DEF LaatikonPyoriminen OrientationInterpolator {
  key [ 0.0 0.5 1.0]
  keyValue [
    0.0 0.0 1.0 0.0
    0.0 0.0 1.0 3.14
    0.0 0.0 1.0 6.28
  ]
}

ROUTE Kello.fraction_changed TO LaatikonPyoriminen.set_fraction
ROUTE LaatikonPyoriminen.value_changed TO Objekti.set_rotation

```

Esimerkki 4.3 Laatikon pyörimisen toteuttaminen

4.2.9. Kuvakulmat ja valot

Kuvakulmien määrittämiseen VRML 2.0 -kuvauskielessä on *Viewpoint*-solmu. Mallille voi määritellä yhden tai useamman kuvakulman, joista ensimmäiseksi määritely on aloituskuvakulma. Muita kooditiedostossa valmiiksi määriteltäviä kuvakulmia voi valita ja käyttää katseluohjelmassa. *Viewpoint*-solmussa on määriteltävissä kuvakulmalle sijainti ja katselusuunta. Jos sitä ei määritellä lainkaan, katseluohjelma valitsee aloituskuvakulman. Yleensä katseluohjelma pyrkii valitsemaan paikan positiiviselta z-akselilta siten, että koko malli on näkyvissä. *Viewpoint*-solmussa itsessään ei ole animointiominaisuutta, jolla voisi tehdä mallissa määrättyllä tavalla liikkuvia kuvakulmia [Carey et al., 1997; Vapourtech, 2001].

Erilaisia valotyyppejä VRML 2.0 -kuvauskielessä on yhteensä neljä. *DirectionalLight*-solmulla malliin tehdään erivärisiä suunnattuja valonlähteitä, joiden säteet ovat yhdensuuntaisia solmussa määritellyn suuntavektorin kanssa. *PointLight*-solmu on pistevalo, jolle on määriteltävä sijainti avaruudessa sekä etäisyys jolle valo valaisee. *SpotLight*-solmu on kohdevalo, ja sille on määriteltävä sen sijainti avaruudessa ja suunta mihin se valaisee. Muita

määriteltäviä ominaisuuksia ovat muun muassa säteen leveys ja valaisuetäisyys. *NavigationInfo*-solmussa on lisäksi otsalamppu (engl. headlight), joka on liitetty kuvakulmaan. Otsalamppu valaisee mallissa liikuttaessa suoraan eteenpäin kuten suunnattu valonlähde. [Carey *et al.*, 1997]

4.2.10. Linkitys ja ääni

VRML-mallin objekteista voidaan tehdä linkkejä toiseen tiedostoon, esimerkiksi HTML-tiedostoon, toiseen malliin, tai sen osaan. *Anchor*-ryhmäsolmulla tehdään linkkejä sekä toisiin tiedostoihin että toisiin VRML-malleihin. *Inline*-ryhmäsolmulla voidaan kutsua kooditiedoston ulkopuolisia objekteja ja käyttää niitä mallissa. Ulkopuolisia objekteja varten *Inline*-ryhmäsolmussa on kentät objektin sijainnin ja koon skaalaamiseksi. [Carey *et al.*, 1997]

Äänten tuottamiseen VRML-mallissa on kaksi solmua: *Sound* ja *AudioClip*. *Sound*-solmulla määritellään äänilähteen sijainti mallissa, sekä äänen suunta. Lisäksi solmulle on määriteltävissä kuinka äänen voimakkuus käyttäytyy mallissa liikkujan sijainnin suhteen. Soitettava äänitiedosto määritellään *Sound*-solmun *source*-kentässä joko *AudioClip* tai *MovieTexture*-solmulla. *AudioClip* ja *MovieTexture*-solmussa määritellään soitettavan tiedoston nimi, soiton jatkuva toistaminen, soittonopeus sekä soiton aloitus- ja lopetuskohta äänitiedostossa. [Carey *et al.*, 1997]

VRML 2.0 -kuvauskielessä tuettuja äänitiedostomuotoja ovat WAV ja MIDI. WAV-tiedostot sisältävät digitoitua ääntä, kun taas MIDI-tiedostot sisältävät ainoastaan äänten soittamiseen liittyvät tiedot. MIDI-tiedostot eivät sisällä soitettavaa ääntä vaan tiedot siitä mitä soitinta soitetaan ja millä tavalla. Soittimet ja niiden äänet on sisällytetty tietokoneen äänikorttiin MIDI-standardin vaatimalla tavalla. Tästä syystä WAV-tiedostot ovat huomattavasti suurempia kuin MIDI-tiedostot, joten suurilla WAV-äänitiedostoilla käytettäessä mallin siirtoajat kasvavat. Käytettäessä digitaalisen äänen soittamiseen *MovieTexture*-solmua, on mahdollista käyttää myös MPEG-pakattua digitaalista äänitiedostoa. Tällöin äänitiedoston koko on pienempi, mutta äänen laatu heikkenee häviöllisen tiivistyksen vuoksi. [Carey *et al.*, 1997].

4.2.11. Java ja JavaScript

VRML-mallin toiminnallisuutta on mahdollista lisätä upottamalla siihen Java- ja JavaScript-ohjelmia *Script*-solmun avulla. Tapahtumilla ja ROUTE-rakenteen avulla VRML-malliin saadaan jonkin verran lisää vuorovaikutteisuutta, mutta jos tahdotaan toteuttaa monimutkaisempaa vuorovaikutteisuutta, ovat Java- ja JavaScript-ohjelmat hyvä vaihtoehto. Ohjelmia voi käyttää hyvinkin erilaisiin tehtäviin, mutta useimmiten ohjelma toimii joissakin seuraavista tehtävistä:

- Toisen solmun lähettämien tapahtumien käsittely

- Logiikka ja laskutoimitukset
- Interaktio katseluohjelman kanssa
- Mallin manipulointi (esimerkiksi solmujen lisääminen ja poistaminen)
- Kommunikointi verkon yli esimerkiksi toisen VRML-maailman tai serveriohjelmiston kanssa

Yleisesti voidaan ajatella, että skriptit tarjoavat mahdollisuuden toteuttaa perinteisille ohjelmointikielille ominaisia toimintoja, joita VRML:llä ei pelkästään voida tehdä. [Carey *et al.*, 1997; Vapourtech, 2001]

4.3. OpenGL

4.3.1. Historia

Vuonna 1990 alkunsa saanut OpenGL on suhteellisen nuori ohjelmointirajapinta. Se perustuu Silicon Graphics Inc:n (SGI) [SGI, 2002] kehittämään IRIS GL:ään, joka on 3D-ohjelmointirajapinta SGI:n IRIS-työasemille. IRIS GL ei kuitenkaan ollut helposti siirrettävissä muihin laiteympäristöihin ja SGI:n suunnitelmissa olikin parantaa IRIS GL:n siirrettävyyttä. Vuonna 1992 muodostettiin OpenGL Architecture Board (OpenGL ARB) ohjaamaan OpenGL-rajapinnan kehitystä. Siihen kuuluu jäsenenä useita suuria yrityksiä, kuten Apple, NVIDIA ja Microsoft. [OpenGL ARB, 2002] Kyseisen vuoden heinäkuussa julkaistiin OpenGL 1.0 -rajapinta. SIGGRAPH'92 -konferenssissa järjestettiin tilaisuuksia liittyen OpenGL:ään, ja siellä järjestettiin myös OpenGL ohjelmointikurssi.

OpenGL-rajapinnan seuraava versio 1.1 ilmestyi vuoden 1995 joulukuussa. Vuonna 1996 rajapinnan määrittelyt julkaistiin. Seuraava rajapinnan versio julkaistiin 1998 ja vuonna 2000 OpenGL:n lähdekoodi annettiin julkiseen leviytykseen. Tällä hetkellä uusin versio OpenGL-rajapinnasta on vuonna 2001 ilmestynyt versio 1.3.

4.3.2. Toimintaperiaate

OpenGL [OpenGL, 2002] on rajapinta, joka mahdollistaa kaksi- ja kolmiulotteisten graafisten näkymien luomisen. Se on sekä standardisoitu ohjelmointirajapinta että -rajapinnan toteutus. Toisin sanoen se on joukko funktioita, joilla on laiteriippumattomasti sama syntaksi, ja niiden voi odottaa toimivan samalla tavalla jokaisella OpenGL-rajapintaa tukevalla laitteistolla. OpenGL sisältää noin 120 C-kielellä toteutettua piirtämiseen liittyvää aliohjelmakutsua. Lisäksi suuresta osasta aliohjelmiä on käytettävissä useampi vaihtoehto

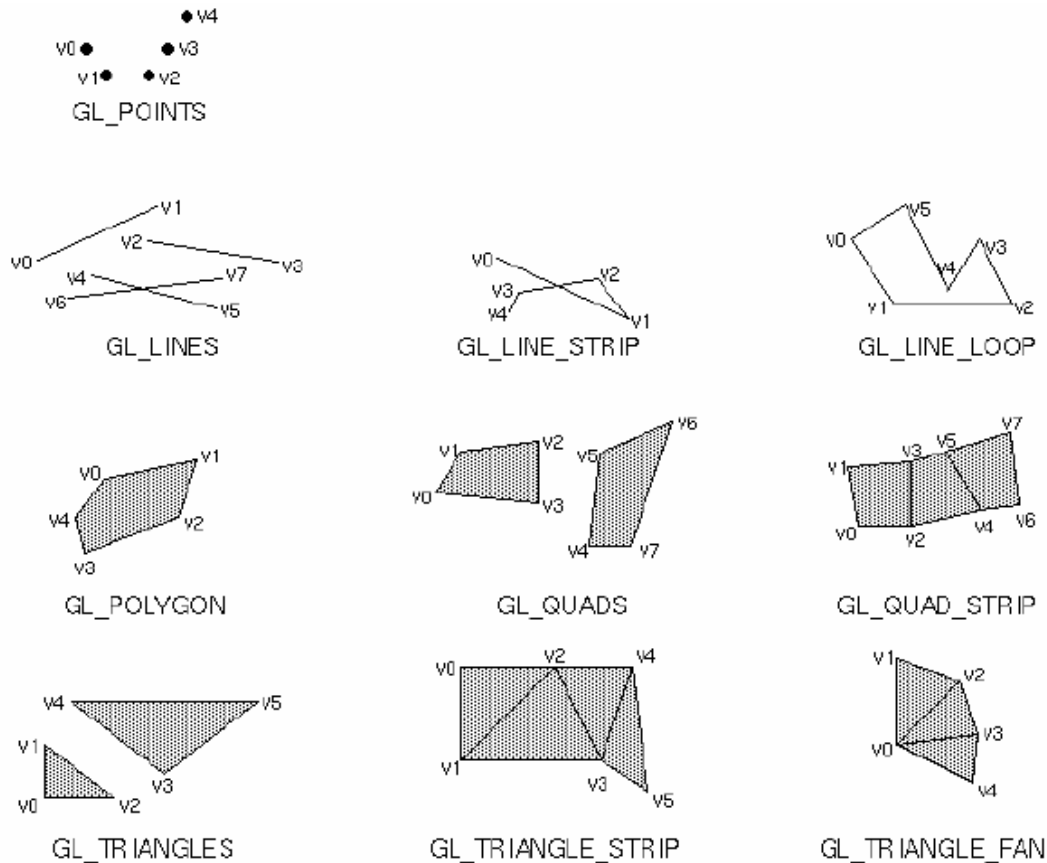
parametrityypeistä riippuen. Lisää aliohjelmaa on lukuisissa OpenGL-rajapinnalle tehdyissä apukirjastoissa, joita esitellään tarkemmin luvussa 4.3.9.

Usein OpenGL-aliohjelmaa kutsuttaessa on mahdollista valita minkä tyyppisiä parametreja aliohjelmalle halutaan välittää. Esimerkiksi kontrolipisteen paikan määrittämiseksi tehty aliohjelma esitellään muodossa $glVertex\{234\}\{s\}i\{f\}d\{v\}(TYPE\ coord)$. Ensimmäisten aaltosulkujen välissä oleva kokonaisluku kertoo, annetaanko piste kaksi-, kolmi- tai neliulotteisessa koordinaatistossa. Toisen aaltosulkuparin sisällä oleva kirjain määrittelee parametrin lukutyyppin (esimerkiksi 16 bittinen kokonaisluku, 32 bittinen kokonaisluku). Aaltosulkujen jälkeen oleva kirjain v mahdollistaa taulukon osoitteen viemisen parametrina. Kaarisulkujen sisällä määritellään aliohjelmalle vietävät parametrit.

OpenGL-rajapinnan rakenne sisältää tilakoneen (engl. state machine), ja sen eri tilojen asettamiseen tarjotaan useita aliohjelmaa. Aliohjelmissa muutettavia tilamuuttujia on rajapinnassa lukuisia. Esimerkiksi väri on yksi tilamuuttuja: kun väri asetetaan johonkin arvoon, tehdään kaikki sen jälkeiset operaatiot asetetulla värillä seuraavaan värin muutokseen asti. Muut valittavat tilat kontrolloivat muun muassa valaistusta, teksturointia ja materiaalin ominaisuuksia. OpenGL on niin sanottu välittömän piirtämisen rajapinta (engl. immediate rendering mode), jossa malli piirretään sitä mukaa kun rajapinnalle syötetään geometrisiä objekteja piirrettäväksi. [Neider *et al.*, 1994]

4.3.3. Objektin rakentaminen

Geometrinen pintojen piirtäminen tapahtuu OpenGL:ssä aliohjelmakutsujen $glBegin(GLenum\ mode)$ ja $glEnd()$ välissä. Kaikki geometriset objektit määritellään joukolla pisteitä, ja pisteet määritellään aliohjelmalla $glVertex\{234\}\{d\}f\{i\}s\{v\}(TYPE\ coords)$. Parametrilla $coords$ määritellään pisteen koordinaatit joko kolmiulotteisessa tai kaksiulotteisessa avaruudessa. Piirrettävän objektin tyyppi määritellään aliohjelman $glBegin(GLenum\ mode)$ parametrilla $mode$. Erilaisia objektityyppejä on OpenGL:ssä useita, mutta peruselementtejä kuten palloja, kartioita tai laatikoita ei ole määritelty, vaan ne on muodostettava käytettävissä olevilla objektityypeillä tai käyttämällä apukirjastoja. Kuvassa 4.3 on esitelty erilaisia tapoja pisteiden yhdistämiseen eri parametreja käyttämällä. Objektia piirrettäessä on määriteltävissä pisteille ja niitä yhdistäville viivoille useita ominaisuuksia, kuten esimerkiksi pisteen koko ja viivan leveys. Viivoille voidaan myös määritellä tapa jolla se piirretään, esimerkiksi onko viiva tyypiltään katkoviiva vai yhtenäinen. Lisäksi voidaan vaikuttaa myös kuviointiin, jolla katkoviiva piirretään. [Neider *et al.*, 1994]



Kuva 4.3. Vaihtoehdot pisteiden yhdistämiseen [Neider *et al.*, 1994]

Esimerkissä 4.4 piirretään laatikko. Ensimmäisellä koodirivillä määritellään millä tavalla pisteet yhdistetään toisiinsa. Tämän jälkeen esitellään monikulmion neljä kulmapistettä. Tämä menettely toistetaan jokaiselle laatikon tahkolle.

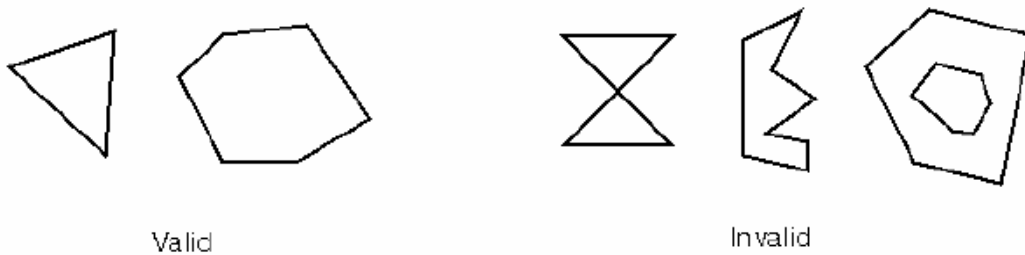
```
//Piirretään laatikon etumainen tahko
glBegin( GL_QUADS );
glVertex3d( 0.0, 0.0, 0.0 ); //Vasen alakulma
glVertex3d( 0.0, 1.0, 0.0 ); //Vasen yläkulma
glVertex3d( 1.0, 0.0, 0.0 ); //Oikea alakulma
glVertex3d( 1.0, 1.0, 0.0 ); //Oikea yläkulma
glEnd();

//Piirretään laatikon loput tahkot
...
```

Esimerkki 4.4 Laatikon piirtäminen

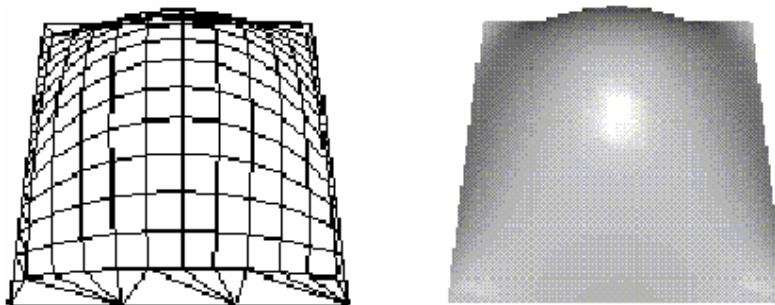
Useista viivoista koostuvien monikulmioiden muodostamisessa täytyy noudattaa tiettyjä sääntöjä. Monikulmioiden on oltava konvekseja, eivätkä esimerkiksi monikulmion reunat saa leikata toisiaan. Kuvassa 4.4 on esimerkkejä

oikein ja väärin tehdyistä monikulmioista. Syynä tällaisiin rajoituksiin on tarkoitus maksimoida piirtämisen nopeus. Yksinkertaisten monikulmioiden piirtäminen ja värittäminen on laskennallisesti huomattavasti nopeampaa ja yksinkertaisempaa kuin vastaavasti monimutkaisten monikulmioiden muodostaminen. Kaikki objektit ovat kuitenkin aina esitettävissä yksinkertaisesti tehdyillä monikulmioilla ja myös monimutkaiset monikulmiot on mahdollista poik-keuksetta paloittaa myös pienempiin osiin. Piirrettäville monikulmioille voidaan määritellä omat ominaisuutensa. Näitä ovat muun muassa määritellyt pinnan teksturoinnille, ja sisä- ja ulkopintojen värittäminen halutuilla väreillä (ks. luku 4.3.5). [Neider *et al.*, 1994]



Kuva 4.4. Oikein ja väärin tehtyjä monikulmioita [Neider *et al.*, 1994]

OpenGL:llä pystyy myös piirtämään kaaria ja kaarevia kolmiulotteisia pintoja, joiden määrittely on usein hankalaa ja monimutkaista. Yksinkertaistetusti ensin kaarelle tai pinnalle määritellään joukko pisteitä ja määritellään niiden avulla piirrettävät pisteet. Määritellyt pisteet sijoitetaan “karttaan”, josta ne sitten myöhemmin piirretään. OpenGL:llä tehtävät kolmiulotteiset pinnat määritellään usein NURBS-pintoina (*Non-Uniform Rational B-spline*), joille on määritetty aliohjelmaa GLUT-kirjastossa (ks. luku 4.3.9). NURBS-pintojen määrittely tapahtuu samalla tavalla kaarien määrittelyn kanssa. Lisäksi voidaan NURBS-pinnalle määritellä halutaanko siitä tehdä rautalankamalli tai pinnoitettu (ks. kuva 4.5). [Neider *et al.*, 1994]



Kuva 4.5. NURBS-pinnan rautalankamalli (vas.) ja pinnoitettu malli (oik.)
[Neider *et al.*, 1994]

4.3.4. Matriisit ja affiinimuunnokset

OpenGL-grafiikassa on kolme erilaista 4×4 -matriisia, joiden muuttaminen vaikuttaa piirtämisen lopputulokseen. Muutettava matriis valitaan aliohjelmakutsulla *glMatrixMode(GLenum mode)*, jolle viedään parametrina joko *GL_MODELVIEW*, *GL_PROJECTION* tai *GL_TEXTURE*. Ensimmäisellä parametrilla vaihtoehdolla viitataan objektin paikan määräävään matriisiin, toisella perspektiiviin ja kolmannella pinnoille tuleviin tekstuureihin. Paikka- ja perspektiivimatriisien käsittelyyn käytetään pinoja. Matriisin lisäämiseksi pinoon ja sieltä poistamiseen käytetään aliohjelmiä *glPushMatrix()* ja *glPopMatrix()*. Aliohjelma *glLoadIdentity()* asetuttaa muutettavan matriisin yksikkömatriisiksi. [Neider *et al.*, 1994; Sun OpenGL]

Matriisien käsittelyyn tarjotaan aliohjelmiä, joiden avulla affiinimuunnokset ovat helppoja tehdä. Objektin tai katselupisteen siirto suoritetaan aliohjelmakutsulla *glTranslate{d f}(TYPE x, TYPE y, TYPE z)*, jossa annetaan siirron suuruus x-, y- ja z-akseleilla parametreina olevilla arvoilla. Kääntämiset tehdään vastaavasti aliohjelmalla *glRotate{d f}(TYPE angle, TYPE x, TYPE y, TYPE z)*. Parametrilla *angle* määritellään käännettävän kulman suuruus ja lopuilla parametreilla suuntavektori, jonka suhteen kääntäminen tapahtuu. Käännettävä kulma annetaan OpenGL:ssä asteina. Objektin kokoa voidaan skaalata kutsulla aliohjelmaa *glScale{d f}(TYPE x, TYPE y, TYPE z)*. Sitä voidaan venyttää antamalla parametreille itseisarvoltaan ykköstä suurempia arvoja. Jos arvot ovat itseisarvoltaan pienempiä kuin yksi, objektia vastaavasti kutistetaan. Jos jokin parametreista on arvoltaan negatiivinen, objekti peilataan kyseisen tason suhteen. [Sun OpenGL]

4.3.5. Värit, varjostaminen ja tekstuurit

OpenGL-ohjelmoinnissa piirrettävän pikselin väriarvo on mahdollista määrätä kahdella eri esitystavalla: RGBA-arvoilla, tai väri-indeksillä (engl. color-index). Väriarvo suljetulta väliltä 0-1 olevalla luvulla, ja piirtoväriin määrittämiseen tarkoitettu aliohjelman kutsu on *glColor{34}{b d f i s ub ui us}v(TYPE red, TYPE green, TYPE blue, TYPE alpha)*. Parametreilla *red*, *green* ja *blue* määritellään kunkin värikomponentin arvo. Parametrilla *alpha* ei ole suoranaista vaikutusta objektin väriin, mutta sitä voidaan käyttää apuna muun muassa objektin reunojen pehmentämiseen ja objektin läpinäkyvyyden luomiseen. Väri-indeksillä puolestaan määrätään yksikäsitteisen lukuarvo jokaiselle värille. Väri-indeksin

määrittämiseen tarkoitettu aliohjelman kutsu on *glIndex{dfs i ub}(TYPE c)*, jossa parametri *c* määrittelee väri-indeksin uuden arvon. [OpenGL Tutorial, 1997; Sun OpenGL]

Monikulmioiden sävyttämiseen on kaksi vaihtoehtoa: tasasävytys ja Gouraud-sävytys. Varjostusmalli valitaan aliohjelman *glShadeModel(Glenum mode)* kutsulla, jossa parametrina välitetään käytettävä varjostusmalli. Jos varjostusmallia ei määritellä, oletusarvona käytetään aina Gouraud-sävytystä. Teksturoidessa objektin pintaa tekstuurilla tarvitaan useita aliohjelmien kutsuja. Lisäksi tekstuurille on määriteltävissä suuri joukko ominaisuuksia, jotka määräävät kuinka tekstuuri käyttäytyy objektin pinnalla. Tekstuurin ominaisuudet määritellään kutsumalla aliohjelmia *glTexEnv{f, i}(GLenum target, GLenum pname, TYPE param)*. Parametrilla *param* määritellään kuinka tekstuuri käyttäytyy monikulmion pintaan mahdollisesti määritellyn värin kanssa - esimerkiksi tekstuurin ja pinnan värit voivat sekoittua lineaarisesti keskenään, tai toinen väreistä voi korvata toisen värin. Parametreille *target* ja *pname* on määriteltävissä ainoastaan yksi vakio muuttujan arvo. Aliohjelmalla *glTexParameter{f, i}(GLenum target, GLenum pname, TYPE param)* määritellään tekstuurille ominaisuuksia, jotka vaikuttavat siihen miten teksturi piirretään monikulmion pinnan päälle. Parametrilla *pname* ilmoitetaan ominaisuus, jota halutaan muuttaa, esimerkiksi tekstuurin pienentämiseen ja suurentamiseen käytettävä funktio, tai tekstuurin reunojen väri. Parametrilla *param* ilmoitetaan määritettävän ominaisuuden uusi arvo. [Sun OpenGL]

4.3.6. Kaksoispuskurointi

OpenGL ei sisällä valmiita aliohjelmakutsuja animaatioiden luomiseksi. Animaatio on toteutettava määrittelemällä objektien piirtokäskyt jokaiselle animaation piirrettävälle kuvalle. Kuvat olisi mahdollista piirtää peräkkäin näytölle siten, että piirretyn kuvan jälkeen piirtoalue tyhjennetään ja seuraava kuva piirretään tilalle. Tällä tekniikalla ei saada aikaan silmälle sulavaa kuvaa pehmeästä animaatiosta, vaan kuva saattaa näyttää sekä nykivältä että välkkyvältä. Lisäksi samaan kuvaan voi piirtyä sekaisin kaksi eri kuvaa. Sulavan animaation tekemiseen käytettävä tekniikka on niin sanottu kaksoispuskurointi (engl. double buffering), jossa piirrettävä kuva piirretään ensin joko ohjelmallisesti tai esimerkiksi tietokoneen näytönohjaimen avulla puskurimuistiin. Kun edellinen kuva on piirretty näytölle, piirretään seuraava kuva puskurista näytölle ja aloitetaan seuraavan piirtäminen puskurimuistiin.

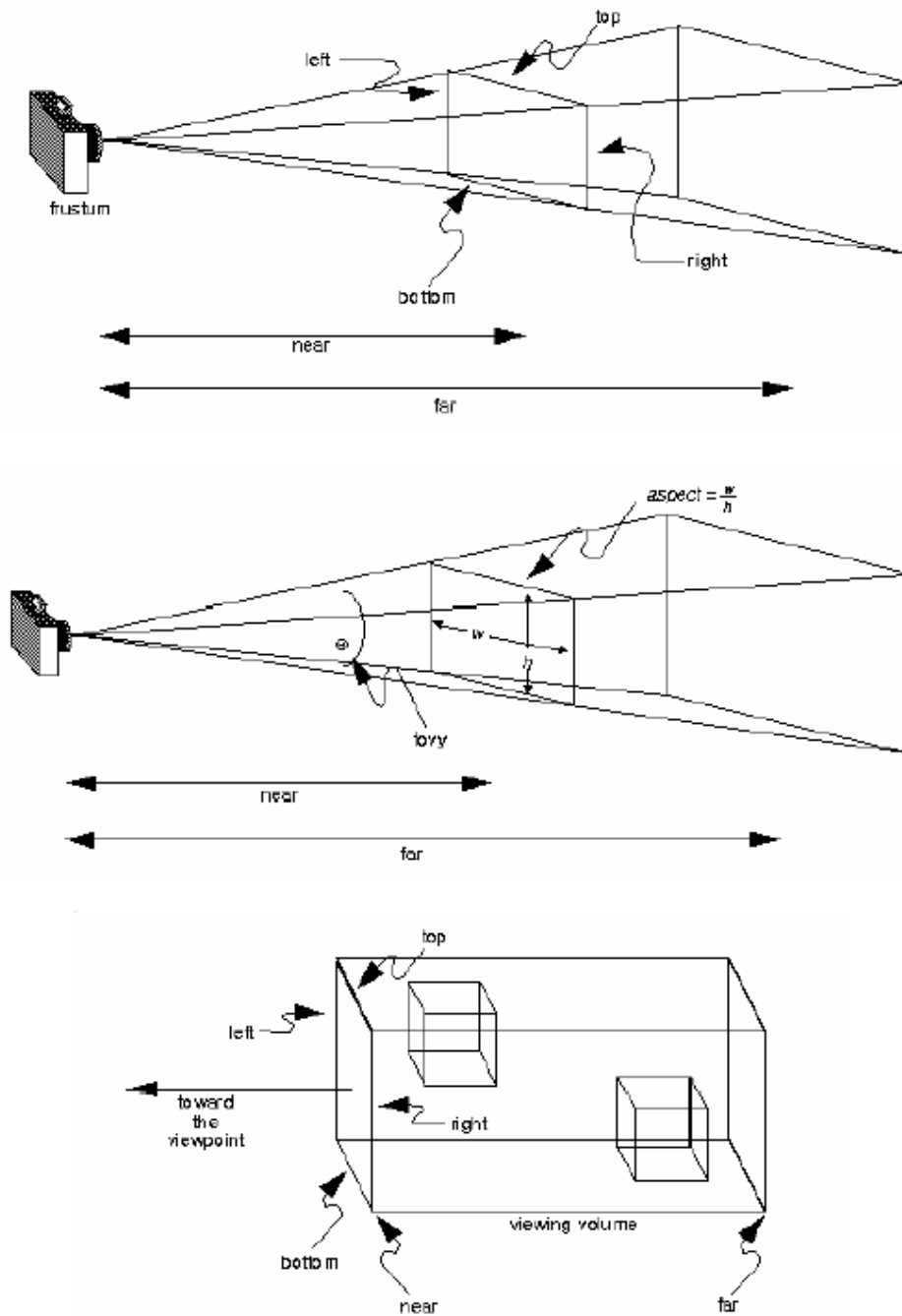
OpenGL-rajapinnassa itsessään ei ole komentoja kaksoispuskuroinnin toteuttamiseksi, koska tämä ominaisuus ei ole välttämättä käytettävissä kaikissa rajapintaa käyttävissä laiteympäristöissä. Kaksoispuskuroinnin käyttö on riippuvainen käytettävästä ikkunointisysteemistä, käyttöjärjestelmästä ja laitteis-

tosta. Kaksoispuskuroinnin toteuttamiseen tarvittavat aliohjelmat löytyvät muun muassa yleisesti käytetystä GLUT-kirjastosta (ks. luku 4.3.9). Puskurin vaihto tehdään kutsumalla GLUT-kirjaston aliohjelmaa *glutSwapBuffers()*. Aliohjelma tekee puskurin vaihdon vaihtaen sillä hetkellä aktiivisena olevan ikkunan taustapuskurin sisällön seuraavaksi piirrettävän puskurin sisällöksi. [Kilgard, 1996]

4.3.7. Mallin katseleminen ja perspektiivi

Katselupisteen valinta ja käyttäminen onnistuu myös ilman OpenGL:n apukirjastoja, mutta yksinkertaisin tapa on käyttää GLU-kirjaston aliohjelmaa *gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)*. Parametreillä *eyex*, *eyey* ja *eyez* määritetään katselupiste avaruudesta josta näkymää katsotaan. Kolme seuraavaa parametria määrittelevät avaruuden pisteen, johon näkymässä katsotaan. Viimeiset kolme parametria määrittelevät katselupisteelle vektorin, joka on suunnattu ylöspäin.

Kolmiulotteisen vaikutelman aikaansaamiseksi täytyy kauempana katselupisteestä olevat objektit piirtää pienemmäksi kuin samankokoiset lähempänä sijaitsevat. Objektien väliset kokoerot riippuvat malliin valitusta perspektiivistä. Kolmiulotteisten näkymän piirtämiseen vaikuttavan perspektiivin määrittämiseen on OpenGL-rajapinnassa useita aliohjelmiä. Yksinkertaisin aliohjelma perspektiivin valintaan on *glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)*. Neljällä ensimmäisellä parametrilla rajataan esitettävä alue (ks. kuva 4.6). Parametrit *near* ja *far* ilmoittavat rajausten etäisyydet katselupisteestä. Rajausten ulkopuolella olevia objekteja ei piirretä näytölle. Monimutkaisempi aliohjelma perspektiivin määrittämiseen on GLU-kirjaston *gluPerspective(double fovy, double aspect, double zNear, double zFar)*. Tällä aliohjelmalla on mahdollista määrittellä parametrilla *fovy* esitettävän alueen katselukulma pystysuunnassa. *Aspect*-parametrilla määritetään näytöllä näkyvän alueen korkeuden ja leveyden välinen suhde. Kahdella viimeisellä parametrilla ilmoitetaan rajausten etäisyydet katselupisteestä. Jos näkymään ei haluta lainkaan perspektiivivaikutelmaa, on mahdollista rajata esitettävä alue aliohjelmalla *gluOrtho(double left, double right, double bottom, double top, double near, double far)* tai kaksiulotteisesti aliohjelmalla *gluOrtho2D(double left, double right, double bottom, double top)*. Neljällä ensimmäisellä parametrilla rajataan esitettävä alue ja loppuilla parametreilla etäisyydet katselupisteestä. Näin on mahdollista käyttää OpenGL:ää myös kaksiulotteisen grafiikan piirtämiseen. [Neider *et al.*, 1994; OpenGL Tutorial, 1997; Sun OpenGL]



Kuva 4.6. `glFrustum(left,right,bottom,top,near,far)` (ylh.),
`gluPerspective(fovy,aspect,zNear,zFar)` (kesk.), `gluOrtho(left, right, bottom,`
`top, near, far)` (alh.) [Neider *et al.*, 1994]

4.3.8. Valot ja pintamateriaalit

OpenGL-rajapinnassa ei ole valmiina perusvalotyyppinä käytettäväksi, kuten esimerkiksi pistevaloa tai suunnattua valonlähdettä. Rajapinnan tarjoamalla

ominaisuuksilla on kuitenkin mahdollista luoda jokainen luvussa 3.3.1 esitellyistä valotyypeistä. Valonlähde määritellään aliohjelmalla *glLight{fv}(GLenum light, GLenum pname, TYPE param)*. Parametri *light* kertoo, mikä valonlähde on kyseessä. Samassa näkymässä voi olla samanaikaisesti ainakin kahdeksan valonlähdettä, ja niihin viitattaessa on mahdollista käyttää vakio-muuttujia LIGHT0...LIGHT8. Käytettävien valonlähteiden lukumäärä riippuu käytettävästä laitteistosta. Parametrilla *pname* määritellään valon ominaisuus, jota ollaan muuttamassa, esimerkiksi valolähteen paikkaa, valon väriä tai valaisusuuntaa. Kolmannella parametrilla *param* määrittää ominaisuuteen liittyvän lukuarvon.

Objektin pintamateriaalin ominaisuudet määritellään samalla periaatteella miten valojen ominaisuudet määritellään. Materiaalista on määriteltävissä väri sekä diffuusit että spekularit heijastusominaisuudet, ja ne määritellään aliohjelmalla *glMaterial{f i}v(GLenum face, GLenum pname, TYPE param)*. Parametrilla *face* ilmoitetaan vaikuttavatko määriteltävät ominaisuudet objektin sisä- tai ulkopinnoille, vai molempiin. Parametrilla *pname* ilmoitetaan mitä materiaaliin ominaisuutta ollaan määrittelemässä, esimerkiksi objektista heijastuva väriä tai objektin kirkkautta. Parametrilla *param* ilmoittaa muutettavan ominaisuuden uuden arvon. [OpenGL Tutorial, 1997; Sun OpenGL] Esimerkissä 4.5 on määritelty piirrettävän objektin pintamateriaalin ominaisuuksia, kuten heijastuvan hajavalon väri ja objektin kirkkausaste, sekä diffuusin ja spekularin heijastuksen väri.

```

GLfloat diffuse_color[] = { 1.0, 0.0, 0.0 };
...
GLfloat shininess[] = { 64.0 };

glMaterialfv( GL_FRONT, GL_AMBIENT, ambient_color );
glMaterialfv( GL_FRONT, GL_DIFFUSE, diffuse_color );
glMaterialfv( GL_FRONT, GL_SPECULAR, specular_color );
glMaterialfv( GL_FRONT, GL_SHININESS, shininess );
...
//Piirretään laatikko
...

```

Esimerkki 4.5 Pintamateriaalin ominaisuuksien määrittely

4.3.9. OpenGL-apukirjastot

The OpenGL Programming Guide Auxiliary Library (GLAUX)

GLAUX-kirjasto kirjoitettiin yksinkertaistamaan teoksen ”OpenGL Programming Guide” esimerkkejä. Kirjastossa on aliohjelmiä ikkunoiden käyttämiseen, syötteiden käsittelyyn ja kolmiulotteisten kappaleiden piirtämiseen (pallo, laatikko, kuutio, sylinteri, kartio). GLAUX-kirjaston aliohjelmat alkavat etuliitteellä *aux*. [Neider *et al.*, 1994]

OpenGL Utility Library (GLU)

GLU on käyttökelpoinen ja OpenGL-ohjelmointia helpottava aliohjelmakirjasto. Se sisältää useita hyödyllisiä rutiineja, jotka käyttävät toteutuksissaan gl-rutiineja. Kirjasto sisältää aliohjelmiä, joita käytetään muun muassa perspektiivin asettamiseen ja matriisien käsittelyyn. Ne eivät suoranaisesti vaikuta piirtämiseen, mutta helpottavat ohjelmointia objektien ja katselupisteiden määrittelyssä. Lisäksi GLU-kirjastossa on aliohjelmiä muutamien geometristen elementtien piirtämiseen (kiekko, pallo, sylinteri). GLU-aliohjelmakirjasto on osa OpenGL-rajapintaa, ja sen aliohjelmat tunnistaa etuliitteestä *glu*. [Segal and Akeley, 1997]

OpenGL Utility Toolkit (GLUT)

GLUT on käytettävän laitteiston ikkunointijärjestelmästä riippumaton työkalukirjasto. Se tarjoaa ohjelmointirajapinnan, joka piilottaa alleen eri ikkunointijärjestelmien monimutkaisuuden ja niiden ohjelmoinnin erilaisuuden. GLUT-kirjasto sisältää rutiineja geometristen elementtien piirtämiseen (pallo, kuutio, kartio ja monikulmaiset kappaleet). Kirjastot myös löytyy aliohjelmiä hiiren ja näppäimistön käyttämiseen sovelluksissa. GLUT ei ole osa OpenGL-rajapintaa, mutta se on asennettavissa useisiin käyttöjärjestelmiin. Kirjaston aliohjelmat tunnistaa etuliitteestä *glut*. [GLUT]

The OpenGL Extension to the X-Window System (GLX)

GLX on suunniteltu koneisiin, jotka käyttävät X-ikkunointijärjestelmää (X Window System). GLX hoitaa lähinnä ikkunoiden alustukset ja avaukset siten, että OpenGL voi suorittaa niihin tarvittavat piirtorutiinit. GLX:n aliohjelmat tunnistaa etuliitteestä *glX*. WGL on vastaava apukirjasto Windows-ympäristöön. [Neider *et al.*, 1994]

Simple DirectMedia Layer (SDL)

SDL on nopea multimediarajapinta äänen ja grafiikan luomiseen, sekä hiiren, näppäimistön ja peliohjaimien syötteiden käsittelyyn. Se on OpenGL yhteensopiva ja se toimii monissa järjestelmissä kuten muun muassa Win32, Linux, BeOS ja MacOS. SDL on ilmainen ja sille on tehty lukuisia apukirjastoja. Sitä on käytetty muun muassa käännettäessä Windows-pelejä Linux-käyttöjärjestelmälle. Kirjaston aliohjelmat tunnistaa etuliitteestä *SDL*. [SDL]

4.4. Java 3D API

4.4.1. Historia

Javan historia alkaa vuodesta 1990, jolloin Sun Microsystems huomasi, että sen asema oli heikentynyt kilpaileviin yrityksiin nähden huomattavasti. Tilannetta parantamaan perustettiin vuonna 1991 uusi huippuohjelmoijista koostuva osasto, joka alkoi kehittää uusia järjestelmiä kulutuselektroniikkaan. Laitteet haluttiin saada ymmärtämään toisiaan. Aluksi laitteissa tarvittavat ohjelmistot oli tarkoitus toteuttaa C++:lla, mutta pian havaittiin, että ohjelmista tulisi liian raskaita eikä niiden siirrettävyys olisi riittävän hyvä. Ongelmien ratkaisemiseksi aloitettiin uuden ohjelmointikielen kehitys, jolle annettiin nimeksi Oak. Uusi kieli perustui C++:aan, mutta se oli laadittu silmälläpitäen keveyttä ja siirrettävyyttä eri laiteympäristöjen välillä.

Vauhtia Oakin kehitykselle antoi WWW:n (World Wide Web) syntyminen vuonna 1993, ja tämä havaittiin oivaksi Oakin kehityssuunnaksi. Oakia kehitettiin siten, että se mahdollisti pienten ohjelmien siirron www-palvelimelta selaimelle, ja ohjelmien suorittamisen selaimella. Ohjelmia kutsuttiin sovelmiksi (engl. applet). Vuonna 1995 Oakin nimi muutettiin Javaksi. [Bank, 1995]

Javan kehitysversio 1.0 julkaistiin tammikuussa 1996, ja sen kehitys oli nopeata. Tällä hetkellä uusin versio on 1.4, joka julkaistiin alkuvuodesta 2002. Java 3D -rajapinnan kehitys aloitettiin vuonna 1995, jolloin Sun ja SGI julkaisivat suunnitelmansa sen kehittämisestä. Java 3D™ 1.1 API oli rajapinnan ensimmäinen versio, ja se julkaistiin vuoden 1998 lopussa. Tällä hetkellä uusin saatavilla oleva versio rajapinnasta on Java 3D™ 1.2.1_04. [Java Timeline, 2000, Java 3D Releases, 2002]

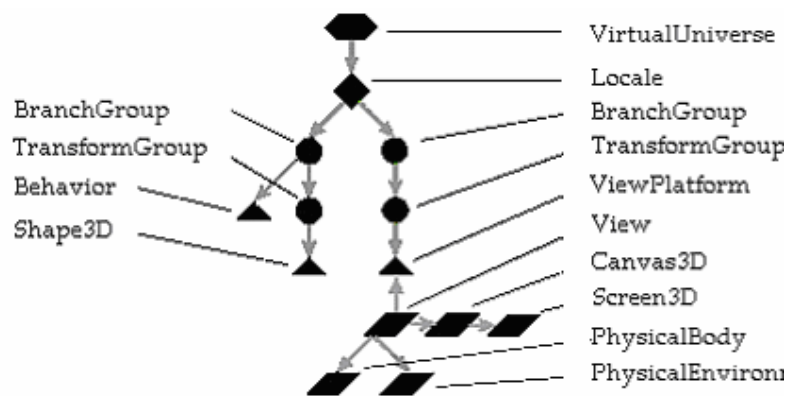
4.4.2. Toimintaperiaate

Java 3D API on luokkakirjasto 3D-grafiikkaa käyttävien ohjelmien ja sovelmien ohjelmointiin. Se tarjoaa korkean tason ohjelmointirajapinnat 3D-grafiikan luomiseen sekä piirtämiseen, ja se toimii OpenGL- tai Direct3D-rajapinnan päällä. Java-ohjelmointikielen luonteen takia Java 3D -ohjelmat ovat ajettavissa useissa eri laiteympäristöissä, ja ohjelmoija voi käyttää kaikkia Javan tarjoamia mahdollisuuksia ja ominaisuuksia. Java 3D ja luvussa 4.2 esitelty VRML lähestyvät 3D-maailman mallinnusta samanlaisella tavalla. Molemmissa malli perustuu hierarkkiseen puurakenteeseen.

Jokainen Java 3D:llä toteutettu ohjelma koostuu ainakin osittain Java 3D:ssä määritellyistä luokkien olioista, se tarjoaa käytettäväksi yhteensä yli 100 erilaista. Suurin osa näistä luokista sijaitsee *java.media.j3d* -pakkauksessa, joka muodostaa Java 3D:n ytimen ja perustan. Muita usein käytettyjä luokkia on lisää *com.sun.j3d.utils* -pakkauksessa, jonka luokat ovat niin sanottuja työkalu-

luokkia (engl. utility classes), jotka rakentuvat usein perusluokkien päälle. Pakkauksesta löytyy työkaluluokkia muun muassa mallin lataamiseen esimerkiksi 3D-mallinnusohjelmista, apuluokkia mallin rakentamiseen, sekä geometrisiä objekteja ja peruselementtejä.

Java 3D:llä toteutettu virtuaalinen malli rakentuu yleensä hierarkkisenä maisemagraafina, joka koostuu Java 3D -luokista luoduista olioista, kuten äänistä, valoista, sijaintitiedoista ja objektien geometrisistä tiedoista sekä ulkoasusta (ks. kuva 4.7.). Kun maisemagraafi on kertaalleen rakennettu, sen sisältämä malli voidaan piirtää yhä uudelleen ilman, että mallia tarvitsee syöttää uudelleen rajapinnalle. Vaikka Java 3D -rajapinnan käytetyin ominaisuus on maisemagraafi, voi rajapintaa käyttää myös välittömän piirtämisen rajapintana (engl. immediate rendering mode). Tällöin voidaan mallia piirtää sitä mukaa kun rajapinnalle syötetään objektien kuvauksia, ja maisemagraafia ei tarvitse käyttää.



Kuva 4.7. Java 3D:llä toteutetun mallin maisemagraafi

4.4.3. Maisemagraafi

Mallin maisemagraafin juurisolmuna on *VirtualUniverse*-luokan olio, jonka tehtävänä on toimia säiliönä (engl. container) kaikille mallien puurakenteille. Lapsisolmuina *VirtualUniverse*-oleksilla on *Locale*-luokan oliot, jotka toimivat säiliöinä *BranchGroup*-solmuolioille. *BranchGroup*-solmuoliot toimivat juurina alipuulle, jossa määritellään esimerkiksi mallin graafisia objekteja tai mallin näkyvyystietoja, joiden perusteella malli piirretään.

Java 3D:n luokkahierarkiassa lähes kaikki luokat perivät *SceneGraphObject*-luokan. Poikkeuksia ovat *VirtualUniverse*- ja *Locale*-luokka. *SceneGraphObject*-luokan kaksi aliluokkaa ovat *Node* ja *NodeComponent*. *Node*-olio on joko *Group*-luokan tai *Leaf*-luokan solmu. *Group*- ja *Leaf*-luokalla on useita aliluokkia. *Group*-luokka on yliluokka, jota käytetään määrittelemään visuaalisten objektien sijaintitietoja mallissa. *Group*-luokan aliluokkia ovat muun muassa

TransformGroup ja *BranchGroup*, joiden ominaisuuksista ja käytöstä kerrotaan tarkemmin luvussa 4.4.5. *Leaf*-luokan perivät luokat määrittelevät muun muassa geometriaa, valaistusta ja ääniä. Näistä luokista kerrotaan tarkemmin luvuissa 4.4.4, 4.4.6 ja 4.4.10. [Bouvier, 2001; Java 3D API, 2000]

4.4.4. Objektin rakentaminen

Geometrinen objektien luomiseksi Java 3D:ssä on *Shape3D*-luokka, jolla kaikki geometriset objektit määritellään. *Shape3D*-oliolle on määriteltävissä erilaisia geometrisiä ominaisuuksia. *Geometry*-luokka on ylliluokka kuvaamaan *Shape3D*-olion geometriaa. *Geometry*-luokan periviä luokkia ovat *CompressedGeometry*, *Raster*, *Text3D* ja *GeometryArray*. *CompressedGeometry*-luokkaa käytetään geometrisen datan (pisteet, kolmiot, viivat) säilyttämiseen pakattuna, *Raster*-luokan avulla voidaan piirtää kaksiulotteinen kuva haluttuun kohtaan mallissa, ja *Text3D*-luokalla on mahdollista malliin määrittellä kolmiulotteinen merkkijono. *GeometryArray*-luokan aliluokat *GeometryStripArray*, *IndexedGeometryArray*, *LineArray*, *PointArray*, *QuadArray* ja *TriangleArray* luovat perustan objektien luomiseen. Niillä määritellään objektiin tarvittavat pisteet, viivat, kolmiot ja nelikulmiot. [Java 3D API, 2000]

Java 3D:n *com.sun.j3d.utils.geometry* -pakkaus (engl. package) tarjoaa valmiita luokkia geometrinen objektien luomiseen. Pakkaus sisältää geometrisistä objekteista *Box*-, *ColorCube*-, *Cone*-, *Cylinder*-, *Sphere*- ja *Text2D*-luokat, joiden avulla malliin voidaan luoda helposti laatikoita, kartioita, sylintereitä, palloja ja kaksiulotteista tekstiä. [Bouvier, 2001] Esimerkissä 4.4 on määritely yksinkertainen laatikko kyseisen pakkauksen avulla.

Appearance-luokalla kontrolloidaan kuinka samassa *Shape3D*-oliossa määritetty geometrinen data esitetään mallissa. *Appearance*-olio ei itsessään määrittele miltä *Shape3D*-olion pitäisi näyttää, vaan se pitää yllä tietoa olioista, jotka määrittävät erilaisia ominaisuuksia ulkoasulle. [Bouvier, 2001] Määriteltäviä ominaisuuksia ovat muun muassa väritykseen liittyvät attribuutit, viivojen ja pisteiden koko, reunanpehmennys, objektin läpinäkyvyys ja pintamateriaali sekä teksturointiin liittyvät ominaisuudet [Java 3D API, 2000]. Esimerkissä 4.5 on esitelty *Appearance*-luokan käyttöä tarkemmin.

```
import com.sun.j3d.utils.geometry.Box;
...
Box laatikko = new Box();
```

Esimerkki 4.6 Laatikon määrittely

4.4.5. Ryhmäsolmut

Ryhmäsolmuilla on ainoastaan yksi isäsolmu, ja ryhmäsolmut voivat koostua useammasta lapsisolmusta. Lapsisolmuja ovat muun muassa ääni-, valo- ja muotosolmut. Kaikki ryhmäsolmut perivät ominaisuutensa *Group*-luokasta. Perittäviä perusominaisuuksia ovat muun muassa lapsisolmujen lisääminen ja poistaminen. *Group*-luokasta periytyviä ryhmäsolmuja on yhteensä kuusi kappaletta, ja jokaisella niistä on lisäksi omat ominaisuutensa sekä niille määritellyt roolit ja tehtävät mallin maisemagraafissa.

BranchGroup-solmu on puurakenteen alipuun juuri, joka on liitettävissä suoraan *Locale*-olioon. *OrderedGroup*-solmun tehtävänä on pitää kirjaa siitä, missä järjestyksessä sen lapsisolmut piirretään mallissa. *OrderedGroup*-solmusta periytyvä *DecalGroup* on tarkoitettu kaksiulotteisen grafiikan piirtämiseen. Esimerkiksi pallon päälle voidaan piirtää tekstiä, tai pinnan päälle voidaan "liimata" tarramainen kuvio. *Switch*-solmulla määritellään mikä tai mitkä siihen kuuluvista lapsisolmuista piirretään malliin.

Samanlaisia muotoja ja ääniä voi olla mallissa useampia kuin yksi. Tällöin on käytännöllistä käyttää jo määriteltyä objektia uudelleen. Tähän tehtävään Java 3D:ssä on *SharedGroup*-solmu, joka toimii juurena kopioitavan objektin puurakenteelle. Se sijaitsee varsinaisen mallin maisemagraafin ulkopuolella, koska sillä ei voi olla isäsolmua. Viittaukset *SharedGroup*-solmuun tehdään *Link*-lehtisolmulla. Jos näin määritellyn objektin ominaisuuksia muutetaan, muuttuvat myös muut vastaavat objektit samalla tavalla.

Mallissa olevien objektien affiinimuunnokset tehdään Java 3D:ssä *TransformGroup*-solmulla. Solmulle tehtävä affiinimuunnos (siirto, kääntö tai skaalaus) ilmoitetaan *Transform3D*-luokan oliolla, ja affiinimuunnos vaikuttaa kaikkiin *TransformGroup*-solmussa oleviin lapsisolmuihin. Java 3D:ssä on käytössä ns. oikean käden koordinaatisto ja etäisyydet mitataan metreinä ja kulmat radiaaneina. Affiinimuunnokset ovat määriteltävissä *Transform3D*-oliolle 4 x 4 -matriiseina tai käyttämällä kullekin muunnokselle määriteltyjä aliohjelmia. [Java 3D API, 2000]

4.4.6. Värit, varjostusmallit ja tekstuurit

Objektin pinnan väriä ja pinnan varjostusta kontrolloidaan *ColoringAttributes*-luokalla. Tämän luokan oliolle annetaan parametreina väri ja sävytysmalli, joilla objektin pinta sävytetään. Väriparametri annetaan *Color3f*-tyyppisellä kolmielementtisellä väri-luokan oliolla, jonka jokaisen värikanavan arvot määritellään liukulukuina välillä 0-1. Värikanavat ovat järjestyksessä punainen, vihreä, sininen (RGB). Mahdollisia vaihtoehtoja objektin pinnan sävytysmalleiksi ovat nopein sävytys (*FASTEST*), laadukain sävytys

(*NICEST*), tasasävytys (*SHADE_FLAT*) ja Gouraud-sävytys (*SHADE_GOURAUD*). Nopeimman ja laadukkaimman sävytyksen määrittely saattaa olla riippuvainen laitteistosta ja käytetystä Java 3D ympäristöstä, mutta yleensä nopein varjostusmalli tarkoittaa käytännössä samaa kuin tasasävytys, ja laadukkain sävytys tarkoittaa Gouraud-sävytystä. [Java 3D API, 2000; Bouvier, 2001] Esimerkissä 4.5 on esimerkki siitä kuinka objektille voidaan määrittellä sen pintamateriaalin ominaisuudet.

Objektin pinnan teksturointiin Java 3D tarjoaa *Texture*- ja *TextureAttributes*-luokat. *Texture*-oliolla määritellään käytettävän tekstuurin ominaisuuksia, joita käytetään objektia ja sen piirrettäessä. Jokaiselle tekstuurioliolle on mahdollista määrittellä muun muassa tekstuuria ympäröivän reunan väri, tekstuurin lähennyksessä ja loitonnuksessa käytettävät suodatinfunktiot ja käytettävään tekstuuritiedostoon liittyvä rakenne. Käytännössä tekstuurien ominaisuuksien määrittelyyn ei käytetä itse *Texture*-luokkaa, vaan sen alaluokkia *Texture2D* ja *Texture3D*. *TextureAttributes*-luokan oliolla määritellään teksturointiin liittyviä ominaisuuksia, joita ovat muun muassa tapa, jolla objektin pinnan väri ja pinnalle tuleva tekstuuri käyttäytyvät keskenään, tekstuurin affiinimuunnokset ja perspektiivin korjaus.

```

Box laatikko = new Box();
Appearance a = new Appearance();
ColoringAttributes ca = new ColoringAttributes(
    new Color3f( 1.0f, 1.0f, 1.0f ),
    ColoringAttributes.NICEST );
a.setColoringAttributes( ca );
a.setMaterial( new Material(
    new Color3f( 0.0f, 0.0f, 0.0f ),
    new Color3f( 0.0f, 0.0f, 0.0f ),
    new Color3f( 1.0f, 0.0f, 0.0f ),
    new Color3f( 1.0f, 1.0f, 1.0f ),
    64.0f ) );
laatikko.setAppearance( a );

```

Esimerkki 4.7 Pintamateriaalin ominaisuuksien määrittely

4.4.7. Animaatio ja vuorovaikutus

Behavior-luokka on abstrakti luokka, jonka avulla toteutettuun malliin voidaan luoda animaatiota ja vuorovaikutteisuutta. Sen tehtävä mallin puurakenteessa on muuttaa maisemagraafin ominaisuuksia ajonaikaisesti. *Behavior*-luokan periviä luokkia ovat niin sanotut interpolaattorit, joiden avulla malliin voi toteuttaa muun muassa animaatiota. *Interpolator*-oliota voi käyttää objektien sijainnin muuttamiseen (*PositionInterpolator* ja *PositionPathInterpolator*), värien vaihtamiseen (*ColorInterpolator*), pyörittämiseen (*RotationInterpolator* ja *RotationPathInterpolator*) sekä läpinäkyvyyden tai koon muuttamiseen (*TransparencyInterpolator* ja *ScaleInterpolator*).

Esimerkissä 4.6 on toteutettu objekti, joka pyörähtää z-akselinsa ympäri kahdessa sekunnissa. Objekti asetetaan ensimmäiseksi *TransformGroup*-solmuun, ja tämän jälkeen määritellään *RotationInterpolator*-oliion tarvitsema pyöritysAlpha-parametri. *Alpha*-oliolla määrittää missä ajassa objektin pyörähtäminen tapahtuu. Lisäksi interpolaattorille viedään parametrina *TransformGroup*-solmu, johon pyörittämisen halutaan kohdistuvan.

Behavior-oliolle on määriteltävissä erilaisia kriteereitä, joiden perusteella se reagoi tapahtumiin. Abstraktilla *WakeupCriterion*-luokalla määriteltäviä kriteerejä ovat muun muassa objektin reagoiminen sen törmätessä toiseen objektiin, ajan kulumisen tai tietyn *TransformGroup*-solmun sisältämän tiedon muuttuminen. Lisäksi Java 3D:n mukana tulevissa pakkauksissa on toteutettuna muutamia *Behavior*-luokan abstraktiota käyttäviä luokkia. Niitä ovat muun muassa näppäimistön käyttöön reagoiva *KeyNavigatorBehavior* ja käyttäjän hiirellä tekemiin toimintoihin reagoivat *MouseRotate*, *MouseTranslate*, *MouseZoom*. Valmiiden luokkien lisäksi malliin on mahdollista ohjelmoida omia *Behavior*-luokkaa käyttäviä luokkia. [Bouvier, 2001; Java 3D API, 2000]

```

TransformGroup objRoot = new TransformGroup();
...
objRoot.addChild( laatikko );
Transform3D zAkseli = new Transform3D();
zAkseli.rotZ( Math.PI/2.0f );
Alpha pyoritysAlpha = new Alpha( -1,
    Alpha.INCREASING_ENABLE, 0, 0, 2000, 0, 0,
    0, 0, 0 );
RotationInterpolator pyorittaja =
    new RotationInterpolator( pyoritysAlpha,
        objRoot, zAkseli, 0.0f, (float) Math.PI*2.0f );
...
objRoot.addChild( pyorittaja );

```

Esimerkki 4.8 Laatikon pyörimisen toteuttaminen

4.4.8. Mallin katseleminen

Mallin katselemiseen tarvittavat ominaisuudet asetetaan Java 3D:ssä *ViewPlatform*- ja *View*-luokan olioilla. *ViewPlatform* on mallin puurakenteen lehtisolmu, joka määrittelee koordinaatiston, johon *View*-olio on liitetty. Se kontrolloi katselijan sijaintia ja asentoa mallin sisällä, ja kun katselija liikkuu mallissa, muutetaan *ViewPlatformin* isäsolmussa olevaa affiniimuunnostietoa. *ViewPlatformille* voidaan määritellä kuinka mallin origo on sijoitettu käyttäjän suhteen; toisin sanoen kuinka se on sijoitettu käyttäjän näkymään. Origo voi asettaa käyttäjän pään kohdalle, jota voi käyttää esimerkiksi varmentamaan, että käyttäjän silmien paikka on virtuaalisessa maailmassa samalla korkeudella

kuin reaali maailmassa. Origon voi asettaa myös käyttäjän jalkoihin tai kuvaruudun keskelle.

View-olio kontrolloi kuinka kolmiulotteinen malli piirretään sen hetkisestä katselupisteestä nähtynä. Se sisältää suuren joukon mallin piirtämiseen vaikuttavia muuttujia sekä aliohjelmia. *View*-oliolle voidaan muun muassa määrittellä onko katseluun käytössä pääripusteinen näyttö (engl. head mounted display) tai kuvaruutunäyttö, mallissa käytettävä projisointi, piirretäänkö käyttäjälle näkymättömiä kappaleita ja tehdäänkö koko mallille reunanpehmennys. Lisäksi *View*-olion avulla saadaan tietoa laitteiston sen hetkisestä suorituskyvystä, kuten esimerkiksi kuinka pitkään viimeisimmän kuvan (engl. frame) piirtäminen kesti. Java 3D -rajapinnan mukana tuleva *com.sun.j3d.utils.universe* -pakkaus lisää käytettäväksi mallin ohjelmoijalle muun muassa *SimpleUniverse*-luokan, jonka avulla on mahdollista luoda nopeasti ja helposti ominaisuuksiltaan minimalistinen ympäristö kolmiulotteiselle mallille. Kyseinen pakkaus sisältää myös helppokäyttöisen *Viewer*-luokan kontrolloimaan *SimpleUniverse*-luokkaa käyttävän mallin katselua. [Java 3D API, 2000]

4.4.9. Valot

Java 3D:ssä on neljä erilaista valotyyppiä, jotka kaikki periytyvät *Light*-luokasta. Jokaiselle valotyypille voidaan asettaa valon väri ja tieto onko valo päällä. *AmbientLight*-luokan avulla on mahdollista luoda malliin taustavalo, joka vaikuttaa mallissa joka paikassa yhtä suurella voimakkuudella. Valon voimakkuuteen ei voi vaikuttaa. Suunnattuja valonlähteitä voidaan luoda *DirectionalLight*-luokasta, ja niiden valaisusuunnat määritetään *Vector3f*-luokan vektorilla. *PointLight*-luokalla on mahdollista tehdä pistevaloja, joille voidaan määrittellä sekä sijainti avaruudessa, että valon voimakkuuden heikkenemisnopeus ja -tapa. Kohdevalojen määrittämiseen rajapinnassa on *Spotlight*-luokka, joka perii ominaisuutensa *PointLight*-luokalta. Kohdevalolle on määriteltävissä valon suunta, valonsäteen kulman suuruus, sekä valon koncentraatio, joka määrittelee kuinka valoteho pienenee siirryttäessä valonlähteestä pois päin.

4.4.10. Äänet

Äänten käyttämiseen tarkoitetut luokat periytyvät *Sound*-luokasta. *Sound*-luokan oliolle on määriteltävissä muun muassa soitettava äänitiedosto, joka määritellään *MediaContainer*-luokan oliolla, soitettavan äänen voimakkuus, soitokertojen lukumäärä, äänen prioriteetti useampia yhtäaikaista ääniä soitettaessa ja äänen soiton kesto. Java 3D:ssä on tuki kolmelle äänitiedostotyyppille: AIF, AU ja WAV.

Rajapinnassa on kolme luokkaa, joiden avulla mallin äänimaailma on mahdollista luoda: *BackgroundSound*, *PointSound* ja *ConeSound*. *BackgroundSound*-luokan oliolla tehdään malliin tehdä taustaääni, joka kuuluu joka paikassa yhtä voimakkaasti ilman että äänen voimakkuus laskisi mallissa liikuttaessa. *PointSound*-luokan oliolla voidaan malliin lisätä äänilähde, joka sijaitsee valitussa avaruuden pisteessä ja lähettää ääntä joka suuntaan yhtä suurella voimakkuudella. *Sound*-luokalta perittyjen ominaisuuksien lisäksi *PointSound*-luokalla on sijainti avaruudessa sekä äänenvoimakkuuden hiljenemiskerroin äänilähteestä poispäin kuljettaessa. *ConeSound*-luokka periytyy *PointSound*-luokasta, ja sillä voidaan tehdä tiettyyn suuntaan ääntä lähettäviä äänilähteitä.

4.4.11. Syöttölaitteet

Java 3D -rajapinnassa ei ole sisäänrakennettua suoraa tukea kaikille syöttölaitteille (engl. input device), mutta sillä on *InputDevice*-rajapinta, jonka voi toteuttaa kommunikoimaan syöttölaitteiden laiteajureiden kanssa. Kolmiulotteiseen maailmaan soveltuvia syöttölaitteita ovat muun muassa peliohjaimet (engl. joystick), erilaiset pallo-ohjaimet ja kypäränäytöt. Rajapinnan ansiosta on mahdollista luoda maailmojen kanssa kommunikoivia syöttölaitteita ilman itse mallin muuttamista.

Rajapinnassa on *Sensor* -luokka kirjaamaan syöttölaitteilta saatavia arvoja, ja sille rekisteröidään tarkkailtava syöttölaite. *Sensor*-olio koostuu sarjasta aika-lemattuja syöttölaitteen antamia arvoja, jotka ilmoittavat muun muassa syöttölaitteen asentoa ja sen nappien sekä kytkimien tiloja. Koska usein mallia käytetään useammalla kuin yhdellä syöttölaitteella, Java 3D pitää kirjata kaikista käytettävistä sensoreista. Sensoreita voi käyttää suoraan koodista, tai sensorit voi tallentaa kolmeen rajapinnassa määriteltyyn kokonaisuuteen: *UserHead*, *DominantHand*, and *NondominantHand*.

Sensorit eivät tallenna kaikkia saamia arvoja, vaan Java 3D pyrkii muokkaamaan mittaustuloksia käyttäjän määrittelemällä tavalla. Määriteltävissä ovat erilaiset tilat muun muassa datahanskoille ja kypäränäytöille. Mittaustuloksista pyritään poistamaan virheelliset tulokset, sekä ennustamaan seuraavia mittaustuloksia, ja täten pienentämään saantiviivettä (engl. latency). [ATK-Sanakirja, 1999; Java 3D API, 2000]

4.5. Muita soveltuvia kieliä ja rajapintoja

Direct3D

Direct3D on osa Microsoftin DirectX-standardia, ja sillä on vankkumaton asema varsinkin Windows-tuoteperheen peleissä ja viihdesovelluksissa. Se on käytännössä syrjäyttänyt monet piirivalmistajien omista rajapinnoista, ja se mahdollistaa ainakin periaatteessa kaikkien Direct3D-sovellusten toimivuuden

Direct3D-yhteensopivien grafiikkapiirien kanssa. Direct3D-rajapinta on käytettävissä myös WindowsCE- ja PocketPC-käyttöjärjestelmää käyttävissä taskutietokoneissa. [Direct3D, 2002]

OpenFlight

OpenFlight on reaaliaikainen 3D-tiedostomuoto, joka on yleisesti käytössä visuaalisessa simulaatiossa. Tiedostomuoto on laitteistosta riippumaton hierarkkinen tietokanta, jossa ovat piirrettävän mallin tiedot. OpenFlight tukee muun muassa piirrettävän grafiikan useita tarkkuustasoja, ääntä, animaatioita, valoja ja materiaaleja. OpenFlight-rajapinta on saatavilla sekä UNIX- että Windows NT -ympäristöille. [OpenFlight, 2002]

Open Inventor

Open Inventor on OpenGL:n päälle rakennettu oliopohjainen rajapinta vuorovaikutteisten 3D-ohjelmistojen tekoon. Se sisältää yksinkertaisen tapahtumiin perustuvan vuorovaikutusmallin, ja sillä voidaan luoda itsenäisesti toimivia animaatio-objekteja. Se määrittelee 3D-tiedostomuodon mallien siirtämiseen sovelluksesta toiseen, ja on ollut myös pohjana VRML-standardille. Open Inventor on laitteisto- ja ikkunointijärjestelmästä riippumaton standardi. [Open Inventor, 2002]

OpenGL Performer

OpenGL Performer on tarkoitettu vaativien vuorovaikutteisten reaaliaikaisten 3D-sovellusten ohjelmointiin IRIX- ja Linux-ympäristössä. Sen tarkoituksena on helpottaa monimutkaisten sovellusten, kuten esimerkiksi keino-odellisuuden ja simulaatioiden ohjelmointia. Performer on rakennettu OpenGL-rajapinnan päälle, ja siinä on tuki useiden suorittimien käyttämiselle. OpenGL Performer tunnettiin aikaisemmin nimellä IRIS Performer, ja vanhemmissa versioissa oli tuki myös IRISGL-rajapinnalle. [OpenGL Performer, 2002]

OpenGVS

OpenGVS on Quantum3D:n markkinoima OpenGL:n päälle rakennettu rajapinta. Se on oliopohjainen ja hyvin pitkälle vastaava kuin OpenGL Performer. OpenGVS tukee useampia eri laitteistoja kuin Performer. Se on optimoitu yhden suorittimen järjestelmiin, ja on täten soveltuva käytettäväksi PC-koneissa. Suosituimmat laiteympäristöt ovat Windows- ja Linux-ympäristöt.

OpenGVS on rakennettu OpenGL-, DirectX- ja Glide-rajapintojen päälle. [OpenGVS, 2002]

Extensible 3D

Web3D 2002 Symposiumissa julkaistu Extensible X3D (X3D) on VRML:n seuraajaksi tarkoitettu kieli, jonka kehitys aloitettiin SIGGRAPH-konferenssissa 1998. X3D soveltuu hajautettujen 3D-sovellusten käyttöön laajemmin kuin VRML, on nopeampi, kevyempi, yhteensopivampi ja vuorovaikutteisempi ja se tukee XML-kieltä (Extensible Markup Language). [X3D, 2002]

5. Ominaisuuksien keskinäinen vertailu

5.1. Käyttöönotto ja käyttäminen

Huomattavin eroavaisuus tässä tutkielmassa käsiteltävien kielien ja ohjelmointirajapintojen välillä ovat niiden rakenteet. VRML on kuvauskieli, jolla kuvataan kolmiulotteisia objekteja, joista koostuvat virtuaalimaailmat ja mallit. Lisäksi mallissa olevien objektien välille voidaan määritellä erityyppisiä suhteita. Objektien kuvaukset luetaan tekstitiedostosta, joiden perusteella VRML-katseluohjelma piirtää mallin. Java 3D - ja OpenGL-rajapinnan käyttäminen virtuaalimaailmojen toteuttamiseen on täysiveristä ohjelmointia. Java 3D rakentuu Java-ohjelmointikielen päälle, ja OpenGL ja sen kirjastot ovat puolestaan kirjoitettu C-ohjelmointikielillä. Tämän takia rajapintojen käyttäjältä vaaditaan ohjelmointitaitoa ennen kuin niiden sisältämien ominaisuuksien käyttäminen onnistuu. VRML:n oppimisen kynnyks on huomattavasti matalampi verrattuna Java- tai C-ohjelmointikielen opetteluun.

Virtuaalimaailmojen rakentaminen näillä ympäristöillä vaatii myös erilaisia työkaluja. VRML-kuvauskielen kirjoittamiseen tarvitaan ainoastaan riittävillä ominaisuuksilla varustettu tekstieditori. Kuitenkin on tavallista käyttää suunnitteluun jotakin 3D-mallinnusohjelmaa, jolla rakennetun mallin voi tallentaa VRML-tekstitiedostona. OpenGL-rajapintaa käytettäessä vaaditaan käyttöjärjestelmään rajapinnan käyttämät kirjastot, jotka ovat usein käyttöjärjestelmässä jo valmiina, tai ne voi kopioida OpenGL-rajapinnan virallisilta verkkosivuilta (<http://www.opengl.org>). Ohjelmakoodin kirjoittamiseen riittää tekstieditori, ja lisäksi tarvitaan kääntäjä, jolla kirjoitettu ohjelmakoodi käännetään ajettavaksi. Kääntäjiä on useita eri käyttöjärjestelmille ja ohjelmointikielille, sekä kaupallisia että ilmaisia. Java- ja Java 3D -ohjelmakoodille on saatavissa ilmainen kääntäjä Sunin verkkosivuilta (<http://java.sun.com>).

Toteutettujen sovellusten käynnistäminen käytettäväksi on myös erilaista eri vaihtoehdoilla toteutettuna. VRML-maailmojen katselemiseen tarvitaan erillinen katseluohjelma, joita on lukuisia eri laiteympäristöille. Osa katseluohjelmista toimii itsenäisinä ohjelmina ja osa toimii osana www-selainohjelmaa. Selainohjelmassa olevalla VRML-katseluohjelmaa voi käyttää esimerkiksi Internetissä olevan VRML-mallin katselemiseen. Java 3D -rajapintaa käyttävien Java-sovellusten käyttämiseen tarvitaan vähintään Java-virtuaalikone (engl. virtual machine), joka tulee kääntäjän mukana. Virtuaalikoneen voi myös kopioida Sunin verkkosivuilta ilman kääntäjää. Java 3D -sovelluksia voi myös käyttää www-selainohjelmalla, jos tietokoneeseen on asennettu Java 3D SDK. OpenGL-rajapintaa käyttävät ohjelmat vaativat koneeseen edellä mainitut

kirjastot. Lisäksi jos tietokoneesta löytyy OpenGL-ominaisuuksilla varustettu näytönohjain, käytetään sen erikoisominaisuuksia grafiikan piirtämisessä.

5.2. Mallin rakentaminen ja sen piirtäminen

Java 3D - ja OpenGL-rajapinnassa ei ole suoraan määriteltyjä geometrisiä peruselementtejä kuten VRML:ssä. Kuitenkin Java 3D:n mukana tulevat pakkaukset ja OpenGL:n useat apukirjastot poistavat tämän puutteen. Objektin rakentaminen on VRML:ssä ja Java 3D:ssä semanttisesti samankaltaista. Objektille voidaan määritellä sen pintamateriaali, materiaalin ominaisuudet ja geometrinen muoto. Samankaltaisuutta on havaittavissa myös objektiryhmien muodostamisessa ja affiinimuunnoksissa. Molemmissa toteutusvaihtoehdoissa malli rakennetaan niin sanottuna maisemagraafina, jonka sisältämät objektit piirretään käymällä graafin hierarkkinen puurakenne rekursiivisesti läpi. Piirtäminen tapahtuu aina tarvittaessa, ja maailman kuvausta ei tarvitse syöttää uudelleen piirtämisestä huolehtivalle rajapinnalle.

OpenGL-ohjelmoinnissa objektien ja mallin rakentaminen eroaa huomattavasti VRML:n ja Java 3D:n tavasta. OpenGL-rajapinta on niin sanottu välittömän piirtämisen rajapinta, jossa mallia piirretään aina sitä mukaa kuin rajapinnalle syötetään objekteja piirrettäväksi. Mallia ei rakenneta muuteltavaan maisemagraafiin, vaan ohjelmoijan tulee itse huolehtia mallinsa vaatimista tietorakenteista, joita käytetään kontrolloimaan mallia ja sen piirtämistä. Tämä voi tuottaa kokemattomalle ohjelmoijalle alussa hankaluuksia. Rakenteiden suunnittelu vaatii huolellisuutta, sillä niitä tarvitaan monissa erilaisissa tilanteissa, kuten esimerkiksi animaatioissa ja sulavassa ruudunpäivityksessä.

VRML:ssä on mahdollista määritellä toisista solmuista ja prototyypeistä uusi solmutyyppi, joka perii solmujen ominaisuudet. Vastaavaa prototyypin luomista ei Java 3D:ssä ole, mutta objektien uudelleenkäyttäminen ja monistaminen onnistuu samalla periaatteella *SharedGroup*-luokkaa käyttämällä. Lisäksi käytössä ovat muut Java-ohjelmointikielen tarjoamat mahdollisuudet. VRML:ssä on DEF- ja USE-avainsanat solmujen monistaminen. OpenGL-rajapinnassa ei ole valmiita ominaisuuksia prototyyppien luomiselle, vaan ohjelmoijan on halutessaan itse suunniteltava ja kirjoitettava vastaaviin ominaisuuksiin tarvittavat ohjelmarakenteet.

Tekstuurien käytössä Java 3D ja OpenGL tarjoavat määriteltäväksi huomattavasti enemmän ominaisuuksia kuin VRML. Objektien pinnoittamiseen voi käyttää VRML:ssä sekä staattista tekstuuria että MPEG-pakattua liikkuvaa kuvaa. Teksturointiin liittyviin erilaisiin ominaisuuksiin ei pysty juurikaan vaikuttamaan. Java 3D ja OpenGL sisältävät laajemmat säätömahdollisuudet liittyen objektin pinnan ja tekstuurin välisiin ominaisuuksiin. Muutettavissa on

useita ominaisuuksia, kuten esimerkiksi suodatinfunktiot, ja tapa, jolla objektin pinnan ja tekstuurin värit sekoittuvat keskenään.

5.3. Valot ja äänet

Sekä Java 3D että VRML tarjoavat samankaltaiset mahdollisuudet mallin valaistamiseen. VRML:ssä on yhteensä neljä erilaista valotyyppiä ja kuten myös Java 3D:ssä. Pistevalo, suunnattu valonlähde ja kohdevalo ovat määritelty molemmissa kielissä, ja Java 3D:ssä on lisäksi mahdollista liittää malliin taustavalon, ja VRML:ssä katselupisteen otsalamppu. Valaistukseen liittyvät ominaisuudet ja tilat ovat molemmissa kielissä hyvin pitkälti samanlaisia. Silti Java 3D:ssä on enemmän mahdollisuuksia valaistuksen ominaisuuksien muuttamiseen ja säätämiseen kuin VRML:ssä. OpenGL:ssä ei ole valmiina samanlaisia valotyyppisiä kuin VRML:ssä ja Java 3D:ssä. Kuitenkin OpenGL-rajapinnan tarjoamalla ominaisuuksilla on mahdollista luoda kaikki edellä mainitut valotyypit.

Ääniä varten VRML tarjoaa yksinkertaiset mutta riittävät ominaisuudet. Käytettävissä on kolme solmutyyppiä, joilla mallin kaikki äänilähteet tehdään. Tuetut tiedostotyypit ovat WAV, MIDI ja MPEG. Java 3D tarjoaa puolestaan huomattavasti monipuolisemmat mahdollisuudet äänten käyttämiseen. Virtuaalimaailmaan on mahdollista laittaa taustääniä, sekä käyttää pistemäisiä ja suunnattuja äänilähteitä. Tuetut tiedostotyypit ovat AIF, AU ja WAV. Säädettäviltä ääniominaisuuksiltaan Java 3D on monipuolisempi kuin vastaavasti VRML.

OpenGL ei sisällä ominaisuuksia äänten luomiseksi, sillä se on ohjelmointirajapinta grafiikan luomiseksi. Ohjelmoijan on käytettävä jotakin OpenGL-apukirjastoa äänten tuomiseksi malleihinsa. Vaihtoehtoja ovat muun muassa luvussa 4.3.9 esitelty Simple DirectMedia Layer -multimediarajapinta, tai turvauduttava käyttöjärjestelmien omiin äänirajapintoihin, kuten esimerkiksi Windows-käyttöjärjestelmissä laajasti käytettyyn DirectSound-rajapintaan.

5.4. Animaatio ja vuorovaikutus

Sekä VRML:ssä että Java 3D:ssä on tarvittavat ominaisuudet animaation ja vuorovaikutuksen toteuttamiseksi. VRML:ssä vuorovaikutuksen luominen perustuu tapahtumien käynnistäjiin ja tapahtumat toisiinsa kytkevään ROUTE-rakenteeseen, joka välittää tapahtuman tiedot kytkennän toiseen päähän. Tapahtumia käynnistäviä sensoreita on VRML:ssä useita. Ne reagoivat muun muassa mallissa tapahtuviin muutoksiin ja käyttäjän tekemiin syötteisiin. Suoraviivaisen animaation tuottamiseen VRML:ssä löytyy niin sanottuja interpolaattoreita, joilla voidaan tehdä malliin muun muassa objektien liikeratoja ja pintavärien vaihteluja.

Java 3D:ssä animaatio ja vuorovaikutus tehdään *Behaviour*-luokasta perittyjen luokkien avulla. *Behaviour*-luokasta perittyjä luokkia ovat interpolaattorit, joilla voidaan vaikuttaa mallin objekteihin samalla tavalla kuin VRML:n interpolaattoreilla. *Behaviour*-luokan olioille on määriteltävissä milloin ja miten ne reagoivat tapahtumiin. Ehtona voi olla esimerkiksi objektien törmäys tai tietyn mittaisen ajan kuluminen. Omia interpolaattoreiden määrittely onnistuu Java 3D:ssä käyttämällä Behavior-luokkaa.

OpenGL ei sisällä samanlaisia monipuolisia ominaisuuksia objektien animaatioiden toteuttamiseksi. Rajapinnassa ei ole aliohjelmaa animaatioiden luomiseksi, ja niiden toteuttaminen on työläämpää, koska OpenGL ei sisällä esimerkiksi samanlaisia tapahtumia havaitsevia sensoreita kuin VRML ja Java 3D. Lisäksi omat haasteensa sijaintiaan tai ominaisuuksia muuttavan objektin luomiseen tuo välittömän piirtämisen rajapinta, jonka takia ohjelmoijan tarvitsee itse toteuttaa ohjelmaansa toimivat rakenteet ongelman ratkaisemiseksi. Sulavan animaation piirtämiseen tarvittavaa kaksoispuuskurointia ei myöskään ole OpenGL:ssä itsessään, mutta se sisältyy muun muassa GLUT-apukirjastoon.

5.5. Syöttölaitteet

Java 3D sisältää *InputDevice*-rajapinnan, jonka toteuttavan luokan voi ohjelmoida kommunikoimaan syöttölaitteen laiteajureiden kanssa. Rajapinnan avulla luotua virtuaalimaailmaa voi kontrolloida monilla erilaisilla syöttölaitteilla. Syöttölaitteilta saatuja arvoja tarkkailemaan Java 3D -rajapinnassa on *Sensor*-luokka, jolla pystyy myös suodattamaan ja käsittelemään syöttölaitteilta saatuja arvoja; virheellisten arvojen poisto on mahdollista.

VRML-kuvauskielessä ja OpenGL-rajapinnassa ei olla määritelty ominaisuuksia syöttölaitteiden tarkkailuun ja käyttämiseen. VRML:ssä syöttölaitteiden tarkkailu ja niiden käyttäminen mallissa on täysin käytettävän katseluohjelman vastuulla. OpenGL:n apukirjastojen avulla voidaan malleihin liittää erilaisten syöttölaitteiden vaatimat ominaisuudet.

5.6. Yhteenveto

Vertailussa olleet ohjelmointirajapinnat ja -kielet eroavat sekä tarjoamiltaan ominaisuuksiltaan että käyttömahdollisuuksiltaan suuresti. VRML-kuvauskielessä on mahdollista toteuttaa virtuaalimaailmoja nopeasti ilman ohjelmointitaitoa. Helpoimmillaan rakentaminen voi olla objektien raahaamista ja sijoittamista VRML-editorilla, ja vaikeimmillaan se on tekstiedoston editointia tekstieditorilla. Java 3D ja OpenGL vaativat puolestaan käyttäjältään ohjelmointitaitoa, ja OpenGL mielellään suurempaa tietämystä tietokonegrafiikasta. Java 3D -rajapinta on rakennettu Java-ohjelmointikielen ja -virtuaalimoottorin päälle. Javan etuna on laiteriippumattomuus, ja Java-tulkkeja onkin toteutettu

useille laiteympäristöille. Java 3D API:a ei kuitenkaan ole toteutettu yhtä monelle laiteympäristölle. Tällä hetkellä tuettavia ympäristöjä ovat ainoastaan Windows ja Solaris. OpenGL:ssä ei ole samanlaista oliomaista lähestymistapaa kuin Java 3D:ssä, vaan ohjelmointi on proseduraalista ohjelmointia. OpenGL-rajapintaa tukevia laiteympäristöjä ovat esimerkiksi Windows, Linux ja Mac OS.

OpenGL on niin sanottu matalan tason grafiikkakirjasto, joka ei sisällä ohjelmointia helpottavia ominaisuuksia kuten VRML ja Java 3D. Sen päälle rakennetaan usein grafiikkamoottoreita ja -rajapintoja, joihin sisällytetään nämä ominaisuudet. Moottoreita ja rajapintoja käyttämällä ohjelmoijan ei välttämättä tarvitse huolehtia kaikista OpenGL-rajapinnan vaatimista asetuksista, eikä hänen tarvitse tuntea OpenGL:n syntaksia tai ominaisuuksia. Java 3D on yksi esimerkki OpenGL:n ominaisuuksia käyttävästä rajapinnasta. Myös useat VRML-katseluohjelmat käyttävät OpenGL-rajapintaa mallin piirtämiseen.

Tarjoamiltaan ominaisuuksilta VRML ja Java 3D ovat lähellä toisiaan. Molemmilla on mahdollista tehdä virtuaalimaailmoja, joissa yhdistyvät muun muassa mallin interaktiivisuus, toiminnallisuus ja elävyys. OpenGL:n avulla on mahdollista tehdä ympäristöjä, joissa käytetään vaativaa grafiikkaa. OpenGL:n heikkoutena ovat siitä puuttuvat ohjelmointia yksinkertaistavat ominaisuudet interaktiivisten ympäristöjen luomisessa. Lisäksi tärkeät ominaisuudet kuten esimerkiksi äänet ja tapahtumiin reagoimiset puuttuvat siitä kokonaan. Näitä puutteita poistamaan on tehty useita OpenGL-yhteensopivia apukirjastoja. OpenGL-rajapinnan suurimpia etuja Java 3D -rajapintaan verrattuna on sen laajempi yhteensopivuus erilaisten laiteympäristöjen kanssa. Lisäksi OpenGL:n ominaisuuksia voidaan kutsua ja käyttää useista eri ohjelmointikielillä toteutetuista sovelluksista. Yhteensopivia ohjelmointikieliä ovat esimerkiksi Ada, C/C++, Fortran, Python, Perl ja Java. VRML:n etuna on sen täysi laiteriippumattomuus. Se toimii jokaisessa käyttöjärjestelmässä, jolle on toteutettu VRML-maailmojen katseluohjelma.

Eroavaisuuksistaan huolimatta VRML, Java 3D ja OpenGL sopivat lähes yhtä hyvin erilaisiin käyttötarkoituksiin. Niissä on yhtä hyvät ominaisuudet toteuttaa malleja, joissa ei ole tarvetta suuremmalle toiminnallisuudelle. Esimerkkinä tästä olisi uuden talon arkkitehtuurin tarkastelu. Eritoten VRML:llä tällaisten mallien toteuttaminen on nopeaa. Monimutkaisinta tällaisten yksinkertaisten mallien toteuttaminen on OpenGL:llä. OpenGL puolestaan soveltuu ylivoimaisesti parhaiten graafisesti raskaiden ja monimutkaisten ohjelmien pohjaksi, joita ovat esimerkiksi kolmiulotteiset suunnitteluohjelmat (CAD) ja tietokonepelit, jotka tällä hetkellä käyttävät laajasti OpenGL-kirjastoa apuna grafiikan piirtämiseen. Java 3D tarjoaa paremmat graafiset ominaisuudet kuin

VRML mutta on teholtaan heikompi kuin OpenGL. Vaikka Java 3D:tä on optimoitu suorituskyvyn maksimoimiseksi, on piirtämisen lisäoptimointi maisema-graafia käytettäessä hankalaa. Hyviä sen tarjoamia ominaisuuksia ovat monipuoliset ääniominaisuudet sekä rajapinta syöttölaitteiden käyttämiselle.

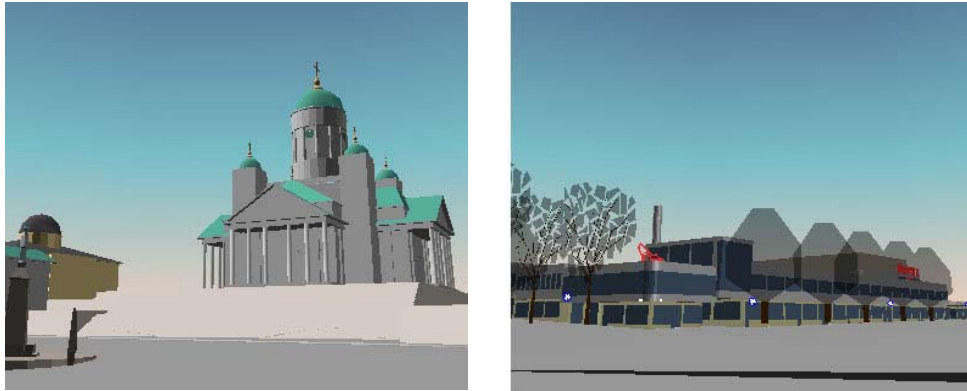
6. Esimerkkejä toteutetuista sovelluksista

6.1. VRML

6.1.1. Virtuaali-Helsinki

Helsinki Arena 2000 -hanke on yhteistyökonsortion vetämä hanke, joka käynnistyi vuonna 1996. Arena-hankkeen tarkoituksena oli tarjota Helsingin asukkailla kehittynyt virtuaalikaupunki, joka tarjoaisi palveluja tietoverkossa. Hankkeen verkkosivuilla (<http://www.arenanet.fi>) on linkkejä erilaisiin palveluihin, kuten esimerkiksi kartta- ja osoitepalveluihin. Osa Arena-hanketta oli luoda kolmiulotteinen malli Helsingistä, jossa voisivat hoitaa asioitaan virtuaalisissa liikkeissä sekä tavata toisia käyttäjiä. [Linturi *et al.*, 2000]

Arena-hankkeen verkkosivuilla on mallinnettuna useita paikkoja Helsingin alueelta VRML-kuvauskielellä (ks. kuva 6.1). Mallit ovat grafiikan yksityiskohdilta kevyitä, mutta paikat ovat tunnistettavissa. Keskustan mallinnus on jaettu useampaan erilliseen tiedostoon, mutta silti latausajat ovat pitkiä jopa nopealla verkkoyhteydellä. Tällä hetkellä Helsinki Arena 2000 -hanke on haudattuna.



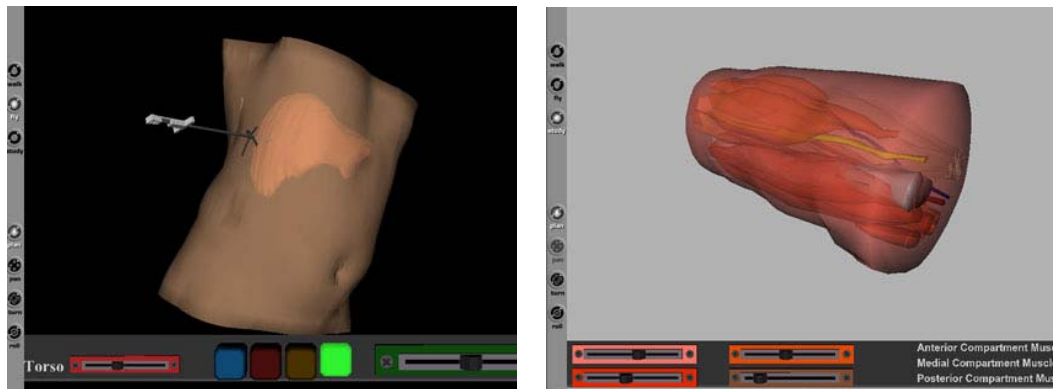
Kuva 6.1. Senaatintori (vas.) ja Lasipalatsi Virtuaali-Helsingissä

6.1.2. Lääketieteelliset sovellukset

VRML on käyttökelpoinen kieli lääketieteellisten sovellusten tekemiseen, sillä mallit on nopeasti luotavissa VRML-kuvauskielellä. Lisäksi mahdollisuus käyttää Java-ohjelmointikieltä sovelluksen osana avaa mahdollisuuksia muun muassa kappaleiden törmäyksistä syntyvien tapahtumien laskemiseksi. [John and Riding, 1999] Manchesterin Yliopisto tarjoaa verkkosivuillaan (<http://synaptic.mvc.mcc.ac.uk/simulators.html>) vapaasti käytettäviä esimerkkejä lääketieteellisistä sovelluksista (ks. kuva 6.2). Sivulla on sovelluksia, joilla

kirurgit voivat harjoitella lääketieteellisiä toimenpiteitä, kuten esimerkiksi koepalan ottamista maksasta, neurokirurgisten toimenpiteiden suorittamista sekä anatomian visuaalista ja vuorovaikutteista opiskelua.

Manchesterin Yliopiston sivuilla olevat sovellukset ovat mielenkiintoisia myös lääketiedettä ymmärtämättömän näkökulmasta. Vaikka henkilökohtaisesti ei ymmärtäisikään toimenpiteiden suorittamisesta mitään, voi vain kuvitella millainen hyöty sovelluksista on esimerkeiksi lääkäreille tai alan opiskelija suorittaville. Hiirtä käyttämällä ohjaus on ainakin aluksi hankalaa ja kömpelöä, mutta tällaiset sovellukset olisivatkin parhaimmillaan datakypäriä ja -hanskaa käytettäessä.



Kuva 6.2. Koepalan ottaminen maksasta (vas.) ja malli reiden lihaksistosta

6.2. OpenGL

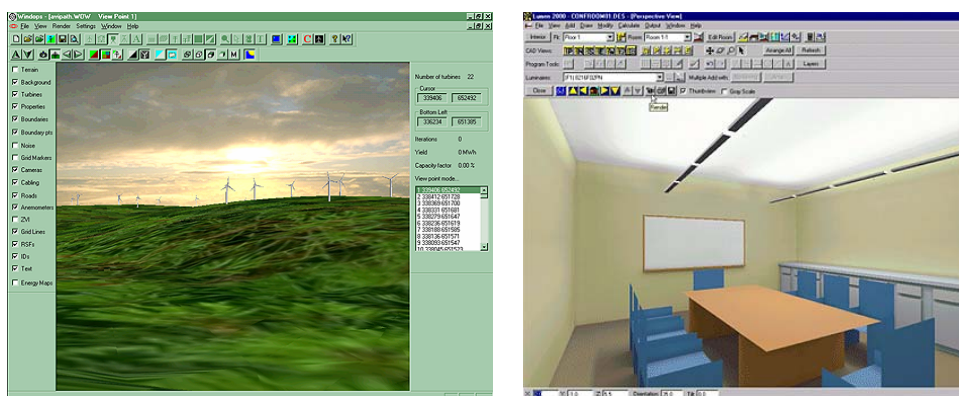
6.2.1. Suunnitteluohjelmat

Erilaiset suunnitteluohjelmat käyttävät grafiikan piirtämiseen OpenGL-rajapintaa. Tyypillisimpiä suunnitteluohjelmia ovat kolmiulotteiset CAD/CAM ohjelmat, joilla voidaan suunnitella ja mallintaa hyvinkin monimutkaisia kolmiulotteisia kappaleita aina ruuvista lentokoneeseen. Lisäksi markkinoilla on suuri joukko suunnittelun tai mallinnuksen eri osa-alueisiin erikoistuneita ohjelmia. Esimerkeiksi suunnittelu- ja mallinnusohjelmista valitsin Ripe Softwaren (<http://www.ripesoftware.com>) suunnittelu- ja mallinnusohjelman WindFarmer, sekä Lighting Technologies Inc:n (<http://www.lighting-technologies.com>) suunnittelu- ja mallinnusohjelman Lumen Micro 2000, jotka molemmat käyttävät piirtämiseen OpenGL-rajapintaa (ks. kuva 6.3).

WindFarmer on ohjelma tuulipuistojen suunnitteluun. Se tarjoaa suunnittelijalle matemaattisia malleja ennustamaan muun muassa tuulivoimayksiköistä ja niiden turbiineista syntyviä meluhaittoja, tai maaston pintamuotojen vaikutuksia. Ohjelman avulla tuulipuistosta voidaan piirtää kolmiulotteisia reaaliaikaisia kuvia, joita voidaan katsella, tallentaa tai tulostaa halutuista ku-

vakulmista. Ohjelmaan on mahdollista tuoda tuulipuiston maa-alueen tiedot esimerkiksi ilmakuvana tai skannattuna karttana.

Lumen Micro 2000 -ohjelmalla voidaan suunnitella valaistusta sekä ulko- että sisätiloihin (ks. kuva 6.3). Suunniteltavat tilat on mahdollista mallintaa käyttämällä ohjelman suunnitteluosiota, jolla voidaan luoda esimerkiksi uutta valaistusta kaipaava neuvotteluhuone. Lisäksi valmiita malleja voi tuoda ohjelmaan muista ohjelmista. Suunnittelua helpottamaan ohjelmassa on valmiiksi mallinnettuna yli 20 000 kappaletta aitoja valaistustarvikkeita käytettäväksi omaa valaistusta suunniteltaessa.



Kuva 6.3. WindFarmer (vas.) ja Lumen Micro 2000

6.2.2. Tietokonepelit

OpenGL-rajapintaa käytetään laajasti piirtämään tietokonepelien monimutkaisia ja usein vaativaa grafiikkaa. Toinen OpenGL-rajapinnan rinnalla käytetty rajapinta tietokonepeleissä on Microsoftin Direct3D, joka on osa Windows-tuoteperhettä. OpenGL-rajapinnan etuna on sen laiteriippumattomuus, joka mahdollistaa suosittujen pelien nopean ja helpon kääntämisen käyttöjärjestelmältä toiselle. Esimerkeiksi OpenGL-rajapintaa käyttävistä tietokonepeleistä olen valinnut id Softwaren (<http://www.idsoftware.com>) vuonna 2001 tekemän pelin Return to Castle Wolfenstein (<http://www.activision.com/games/wolfenstein>) ja Valve Softwaren (<http://www.valvesoftware.com>) vuonna 1998 tekemän pelin Half-Life (<http://half-life.sierra.com>) (ks. kuva 6.4).

Return to Castle Wolfenstein käyttää grafiikan piirtämiseen id Softwaren tekemän Quake III:n (<http://www.activision.com/games/quake3>) pelimoottoria, joka on tehty OpenGL-rajapinnan päälle. Moottorin ovat lisensoineet käyttöönsä myös monet muut pelitalot, ja Quake III:n pelimoottoria pidetäänkin yleisesti yhtenä parhaista pelimoottoreista pelikäyttöön, ja käytettäväksi modernien 3D-kiihdytettyjen näytönohjaimien kanssa.

Half-Life julkaistiin vuonna 1998, ja siitä tuli heti yksi eniten kautta aikojen myydyimmistä tietokonepeleistä. Pelin grafiikka oli ennennäkemätöntä, ja varsinkin ihmishahmojen mallinnus oli useita askeleita edellä muita tietokonepelejä. Half-Life on voittanut julkaisemisensa jälkeen yli 50 pelialan palkintoa ympäri maailman. Käyttäjä pystyy valitsemaan grafiikan piirtämiseen käytetyn rajapinnan OpenGL:n tai Direct3D:n väliltä Windows-versiossa. Muissa laiteympäristöissä (Linux, MacOS) piirtämiseen käytetään OpenGL-rajapintaa.

Pelinä Half-Life on säilyttänyt edelleen asemansa uusien pelien puristuksessa. Siitä on tehty lukuisia modifikaatioita, joista tunnetuimpana esimerkkinä on Counter-Strike (<http://www.counter-strike.net>) - verkossa moninpelattava joukkuepeli, jolla on pelkästään Suomessa tuhansia pelaajia. Half-Lifen grafiikkamoottori tuottaa edelleen laadukasta grafiikkaa, joka osaltaan on auttanut pelin säilymistä suosittuna vielä useita vuosia sen julkaisemisen jälkeen.



Kuva 6.4. Return to Castle Wolfenstein (vas.) ja Half-Life

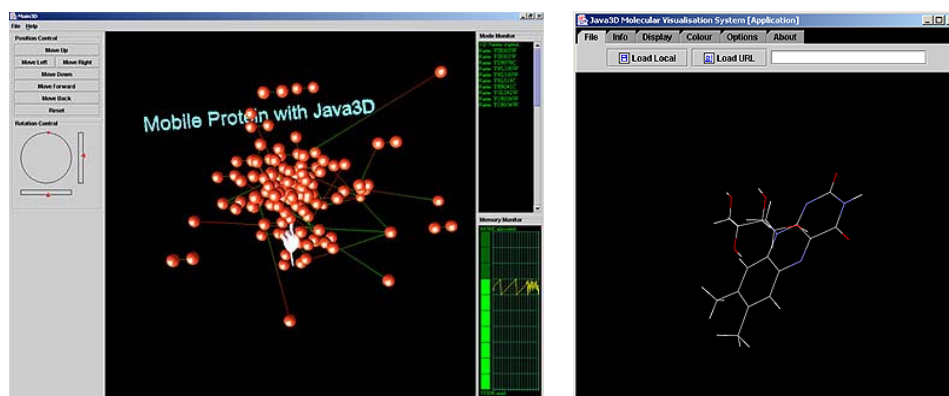
6.3. Java 3D

6.3.1. Kolmiulotteisten mallien katseluohjelmat

Yksi Java 3D:n suosituimmista käyttökohteista on sen käyttäminen erilaisten kolmiulotteisten katseluohjelmien näkymien piirtämiseen. Tämä on selitettävissä Java 3D:n monipuolisilla ja helppokäyttöisillä ominaisuuksilla, joilla on nopeaa toteuttaa monimutkaisemmankin mallin esittäminen. Lisäksi Java 3D:ssä on sisäänrakennettuna tuki esimerkiksi VRML-tiedostotyyppisille malleille, joiden hyödyntäminen on tästä syystä erittäin käytettyä Java 3D - pohjaisissa ohjelmissa.

Esimerkeiksi kolmiulotteisten mallien katseluohjelmista olen valinnut ohjelmat "Mobile" of Protein Interactions (<http://www.cs.uga.edu/~hui/>

cs6900/final.html) ja Java3D Molecular Visualisation System (<http://www.adcworks.com/ContentManager.jsp?cid=31>) (ks. kuva 6.5). Molemmat ohjelmat ovat idealtaan ja toteutukseltaan samankaltaisia. ”Mobile” of Protein Interactions -ohjelmassa (MPI) käyttäjä voi katsella eri proteiinien välistä vuorovaikutusta. Mallissa pallot kuvaavat proteiinia, ja palloja yhdistävät suorat proteiinien välistä vuorovaikutussuhdetta. Käyttäjällä voi valita halutun proteiinin ja saada tällöin lisätietoa proteiinista. Mallia on mahdollista loitontaa tai lähentää sekä liikuttaa. Java3D Molecular Visualisation System on puolestaan ohjelma, joka mallintaa molekyyliä kolmiulotteisesti. Muuten se on toiminnoiltaan samanlainen kuin proteiineja mallintava MPI.



Kuva 6.5. ”Mobile” of Protein Interactions (vas.) ja Java3D Molecular Visualisation System

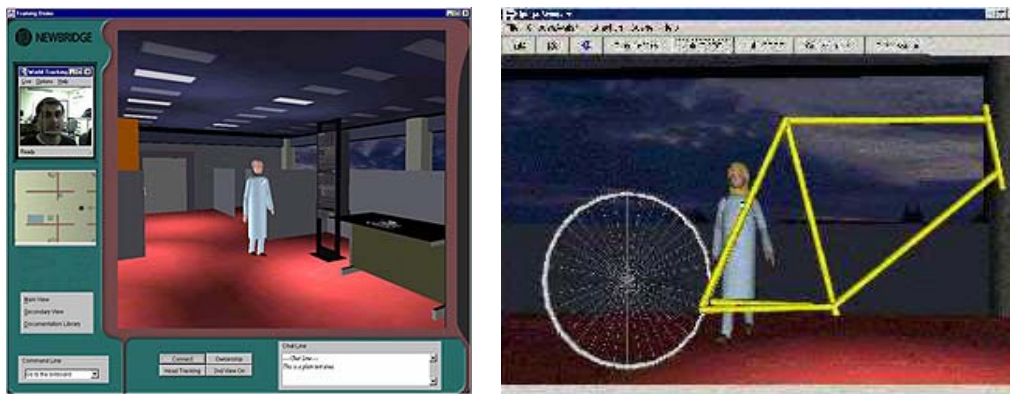
6.3.2. COVET ja COSMOS2

Java 3D:n piirto-ominaisuuksia voidaan käyttää myös osana suurempaa kokonaisuutta, jossa on käytössä useita muita ohjelmointikieliä. Esimerkkeinä tästä ovat COVET (Collaborative Virtual Environment for Training) ja COSMOS2 (Collaborative System framework based on MPEG4 Objects and Streams), jotka ovat virtuaalisia ympäristöjä työtehtävien harjoitteluun. Ohjelmat on suunniteltu vähentämään koulutukseen kuluihin menojen määrää, sillä työntekijöiden koulutukseen kuluvat resurssit ovat monille teollisuuden aloille suuri menoerä. Usein on myös mahdollista, että kouluttaja ja koulutettava ovat maantieteellisesti hyvinkin kaukana toisistaan. Virtuaalisessa koulutusympäristössä voivat sekä koulutettava että kouluttaja toimia yhdessä ja harjoitella tulevia työtehtäviä ennakkoon, ja tehdä riittäviä määriä toistoja ilman, että koulutettava ja kouluttaja ovat fyysisesti samassa tilassa.

COVET on prototyyppi, jolla voidaan toteuttaa yksinkertaisia harjoittelutehtäviä. Sillä on mahdollista harjoitella hajonneen laitteen osan vaihtamista, mutta tehtävä on helposti muunneltavissa erilaiseksi. Tehtävän aikana

kouluttaja ohjaa tehtävän suorittajaa, ja opastaa kuinka oikeaoppinen suoritus tehdään. COVET on toteutettu pitkälti Java-ohjelmointikielellä ja näkymän piirrosta huolehtii Java 3D. Lisäksi ohjelman toteutukseen on käytetty myös C++-ohjelmointikieltä, jolla on toteutettu muun muassa ohjelman vaatima verkkokommunikaatio ja käyttäjän päässä olevan näytön ohjaaminen. Java Native Interfacea (JNI) on käytetty eri ohjelmointikielillä toteutettujen komponenttien väliseen kommunikointiin. [de Oliveira *et al.*, 2000; Hosseini and Georganas, 2001]

COSMOS2-projektin tarkoituksena on kehittää ohjelmistokehys (engl. framework), jota voidaan käyttää toteutettaessa virtuaalisia harjoitteluympäristöjä. Ohjelmistokehys on toteutettu Java-ohjelmointikielellä, ja näkymän piirtämisestä vastaa Java 3D. Ohjelman on tarkoitus toimia usealla yhtä-aikaisella käyttäjällä, jotka voivat halutessaan ladata ympäristöön uusia objekteja ja käyttää niitä. Kuvassa 6.6 käyttäjät kokoavat yhdessä COSMOS2-ympäristössä osista polkupyörää. [Hosseini and Georganas, 2001]



Kuva 6.6. COVET (vas.) ja COSMOS2 [Hosseini and Georganas, 2001]

7. Esimerkkisovellusten toteuttaminen

7.1. Toteutetut sovellukset

Virtuaalimaailmojen toteuttamista kokeilin käytännössä jokaisella tässä tutkimuksessa käsitellyllä vaihtoehdolla. Esimerkeillä on tarkoitus esitellä luvussa 5 läpikäytyjä ominaisuuksia ja havainnollistaa sovellusten toteutusprosessia toteutukseen valituilla ohjelmointiympäristöillä. Sovellukset pyrittiin toteuttamaan eri toteutustavoilla mahdollisimman yhteneväisiksi, sillä niitä käytettiin luvussa 9 käsiteltävässä piirtonopeustestissä. Tästä syystä sovellutuksista jätettiin pois esimerkiksi VRML:stä ja Java 3D:stä löytyvät ääniominaisuudet, sillä OpenGL-sovelluksissa olisi jouduttu turvautumaan apukirjastoihin äänen tuottamiseksi. Lisäksi esimerkkisovelluksissa ei käytetä grafiikan piirtoon liittyviä optimointia, kuten esimerkiksi yksityiskohtien karsintaa tai algoritmeja piilopintojen poistamiseksi. [Koikkariinen *et al.*, 2001] Sovellukset toteutettiin Windows-ympäristössä, ja sovellusten toteuttamista muissa laiteympäristöissä ei käsitellä.

Toteutetut sovellukset ovat pyöriä bentseenimolekyylejä piirtävä BenzeneDemo, aurinkokuntamme planeettojen liikettä simuloiva SolarSystemDemo ja pyörivää tunnelia piirtävä TunnelDemo (ks. kuva 7.1). BenzeneDemo-sovelluksessa on yhdeksän kolmiulotteista bentseenimolekyyliä, jotka pyörivät akselinsa ympäri. TunnelDemo-sovelluksessa käyttäjä on pituusakselinsa ympäri pyörivän kahdeksankulmaisen tunnelin sisällä. SolarSystemDemo-sovelluksessa on mallinnettu aurinkokuntamme planeettojen liikkeitä. Planeettojen radat eivät ole todellisia, koska simulaatiossa ei ole mallinnettu painovoiman vaikutuksia. Planeettojen kiertoajat, koot ja kiertoratojen säteet ovat suhteutettuja todellisista arvoista. Yksi sekunti simulaatiossa vastaa noin kymmentä Maan vuorokautta. Käyttäjällä voi liikkua ja valita katselupisteensä vapaasti jokaisessa toteutetussa sovelluksessa.



Kuva 7.1. BenzeneDemo (vas.), SolarSystemDemo ja TunnelDemo

7.2. Toteutusprosessit

7.2.1. VRML

VRML-mallien tekeminen käytettävät ohjelmat voidaan jakaa karkeasti kahteen ryhmään: tekstieditoreihin tai kolmiulotteisiin mallinnusohjelmiin. Tarjolla on molemmissa ryhmissä useita erilaisia ohjelmia, joissa on erilaisia suunnittelua ja mallin laadintaa tukevia ominaisuuksia. Esimerkki monipuolisesta tekstieditorista on ParallelGraphicsin VrmlPad 2.0 (<http://www.parallelgraphics.com/products/vrmlpad>). Esimerkiksi ilmaisesta kolmiulotteisesta mallinnusohjelmasta löysin Stephen F. Whiten kehittämän white_dune (<http://www.csv.ica.uni-stuttgart.de/vrml/dune/>), jolla mallin luominen tehdään sijoittamalla ja muokkaamalla objekteja suoraan mallin sisällä. Kuitenkin white_dune vaikutti vielä sen verran keskeneräiseltä ja epävakaalta ohjelmalta, että en käyttänyt sitä esimerkkisovellusten rakentamiseen.

VrmlPadista on saatavissa ilmainen kokeiluversio, jonka käyttö ja ominaisuudet ovat rajoittuneimpia maksulliseen versioon verrattaessa. VrmlPadin hyvä ominaisuus on kirjoitetun koodin syntaksin ja rakenteen tarkastaminen. Ominaisuuden avulla on mahdollista huomata nopeasti koodissa olevat virheet. Lisäksi hyödyllinen VrmlPadin ominaisuus on koodin täydentäminen. Ohjelmassa sisältää myös useita muita käytännöllisiä ominaisuuksia, muun muassa eri rakenteiden värikoodaus, objektien pintamateriaalien muokkaaminen visuaalisesti sekä kirjoitetun VRML-tiedoston siirtäminen esimerkiksi Internetiin.

Mallien kirjoittaminen oli VrmlPadin avulla nopeaa. Ohjelman ominaisuuksien avulla virheiden tekeminen oli vähäistä, ja virheelliset kohdat huomasi nopeasti. Sovelluksista BenzeneDemo ja TunnelDemo olivat yksinkertaisia ja suoraviivaisia toteuttaa, aikaa kului eniten lähinnä objektien suunnitteluun ja niiden kasaamiseen. Tässä vaiheessa olisi ollut erityisesti hyödyllistä, jos ohjelmassa olisi ollut kolmiulotteinen näkymä malliin, mutta tämän puutteen pystyi korjaamaan käyttämällä VRML-katseluohjelmaa editorin rinnalla. Animaation tekeminen kaikkiin malleihin oli ROUTE-rakenteen avulla yllättävän yksinkertaista. Samoin oli myös teksturoitujen pintojen tekeminen ja pintamateriaalien ominaisuuksien muuttaminen halutunlaiseksi. SolarSystemDemo-sovelluksen jaoin useampaan eri tiedostoon käyttämällä *Inline*-ryhmäsolmua tiedon hajauttamiseen. Useampaan tiedostoon siirtyminen teki sovelluksen hallinnasta huomattavasti selkeämpää ja helpompaa.

7.2.2. OpenGL

OpenGL-rajapinnan käyttöönottoaminen on Windows-käyttöjärjestelmässä yksinkertaista, sillä lähes jokaisessa käyttöjärjestelmän eri versiossa on OpenGL-rajapinnan käyttämät tiedostot. Jos tietokoneessa ei ole OpenGL-yhteensopivaa näytönohjainta, niin tiedostojen avulla piirretään mallit ohjelmistopohjaisesti. Näytönohjaimen käyttämät OpenGL-laiteajurit tulevat näytönohjaimen mukana, tai ne saa kopioitua joko näytönohjaimen tai grafiikkapiirin valmistajan verkkosivuilta. OpenGL-yhteensopivan näytönohjaimen käyttö on suositeltavaa, sillä se piirtää OpenGL-grafiikkaa nopeammin kuin ohjelmistopohjaista piirtämistä käyttävä laitteisto. Luvussa 7.3.3 esitellään tarkemmin ohjelmistopohjaisen piirron ja 3D-näytönohjaimen välistä nopeuseroa OpenGL-grafiikkaa piirrettäessä. Rajapinnan ohjelmointiin tarvitaan jokin OpenGL-yhteensopiva ohjelmointikieli ja sitä käyttävä ohjelmointiympäristö, jossa on mukana ohjelmakoodin kääntämiseen tarvittava kääntäjä. Lisäksi kääntäjään pitää asentaa kääntämisessä tarvittavat OpenGL-kirjastot ja apukirjastot jos niitä ei tule ohjelmointiympäristön mukana.

Kääntäjiä on useita eri laiteympäristöihin, valittavissa on joko ilmaisia tai kaupallisia versioita. Ilmainen OpenGL-ohjelmointiin sopiva C++-kääntäjä on gcc (<http://gcc.gnu.org>), josta on versiot muun muassa Windows- ja Linux-käyttöjärjestelmälle. Tässä tutkielmassa esiteltävät OpenGL-ohjelmointiesimerkit ohjelmoitiin kaupallisella Microsoft Visual C++ 6.0 -ohjelmointiympäristöllä (<http://msdn.microsoft.com/visualc/>), joka sisältää kääntäjän ja tekstieditorin. Ohjelmointiympäristön mukana tulee myös OpenGL-rajapinnan tarvitsemat kirjastot ja tiedostot. OpenGL-apukirjastoja ei sen mukana tule, mutta ne voi yleensä kopioida ilmaiseksi kyseisen apukirjaston verkkosivuilta. OpenGL-rajapinnan virallinen verkkosivu (<http://www.opengl.org>) ylläpitää listaa eri apukirjastoista ja niiden verkkosivuista.

Esimerkkisovellusten ohjelmointi aloitettiin luomalla projekti merkkipohjaiselle sovellukselle. Esimerkkien ikkunointi ja syöttölaitteiden käyttö toteutettiin GLUT-apukirjaston avulla, joka piti asentaa ohjelmointiympäristöön. Kopioidussa paketissa tuli mukana kaikki apukirjaston käyttämiseen tarvittavat tiedostot, kuten apukirjaston otsikkotiedosto (engl. header file), linkittämiseen tarvittava kirjasto ja käännettyjen ohjelmien vaatima glut32.dll-kirjasto. Paketin mukana tulevien ohjeiden avulla pystyi ohjelmointiympäristön asetukset muuttamaan nopeasti toimintakuntoon.

Ennen ohjelmoinnin aloittamista suunnittelin esimerkeille yhteisen rakenteen, joka olisi mahdollisimman toimiva. GLUT-aliohjelmilla hoidettiin ikkunan luominen, mallin piirtämiseen käytettävän aliohjelman rekisteröinti ja näppäimistön tarkkailu. Näppäimistön tarkkailuun rekisteröin oman aliohjel-

mansa. Muilla esimerkkeihin toteuttamillani aliohjelmilla ohjattiin mallin valaistusominaisuuksia, perspektiivin asetuksia ja tekstuurien käyttöä. Rajapinnan yleisille asetuksille, kuten valaistukselle, toteutin omat aliohjelmansa. Teksturointiin käytettävien BMP-kuvatiedostojen lataamiseen käytin Jacob Marnerin tekemää ohjelmaa (<http://www.rolemaker.dk/nonRoleMaker/loader/>). Toteutin esimerkkeihin lisäksi komponentit ruudunpäivitysnopeuden (fps) laskemiseen ja tulostamiseen (FPSCounter- ja FPSLogger). Java 3D -esimerkkeihin toteutin myös samoja tehtäviä hoitavat komponentit.

Mallin toteuttaminen oli ensimmäisellä kerralla paljon hankalampaa kuin VRML-kuvauskielellä tai Java 3D -rajapinnalla. Koska tein ensimmäiseksi TunnelDemo-esimerkin, törmäsin heti objektin muodostamisen vaikeuteen. Tunnelin seinissä käytettyjen laatikkojen tekeminen vei käsittämättömän paljon aikaa. Mutta esimerkkien tutkimisella ongelmat ratkesivat. Yksi tutustumisen arvoisista verkkosivuista on GameDev.net (<http://www.gamedev.net>), joka sisältää runsaasti kommentoituja esimerkkejä juuri aloitteleville OpenGL-ohjelmoijille. Aiheittain verkkosivulle ryhmiteltyjen esimerkkien tutkimisella avautuivat myös monet muut vastaavanlaiset ongelmat. GLU-apukirjastossa olevat aliohjelmat pallon ja sylinterin piirtämiseen helpottivat muiden esimerkkien ohjelmointia.

OpenGL-rajapinnan kanssa törmäsin pariin ratkaisua vaativaan ongelmaan. Toinen niistä oli mallissa liikkuminen, johon en löytynyt haluamaani ratkaisua. Suunnitelmissani oli toteuttaa mallissa liikkuminen VRML-katseluohjelmissa käytetyllä tavalla, jossa kameran siirtäminen on koordinaattiakseleista vapaata. Koska ongelmaan ei löytynyt riittävän hyvää ratkaisua, toteutin yksinkertaisen ratkaisun, jossa liikkuminen on sidottu koordinaattiakseleihin. Käyttäjä voi liikuttaa näppäimistöllä kameraa ainoastaan pitkin x-, y- ja z-akselia, ja kääntää samalla mallia haluamaansa asentoon eri akselien suhteen. Ratkaisu ei ole kovin käytännöllinen, mutta riittävä toteutetuissa esimerkeissä liikkumiseen.

Toinen ratkaisua vaativa ongelma koski animaatioiden tekemistä. OpenGL ei sisällä VRML:stä ja Java 3D:stä tuttuja interpolaattoreita, joilla voidaan tarkasti määrätä esimerkiksi kappaleen pyörähtämiseen kuluva aika. Koska ruudunpäivitysnopeus on riippuvainen käytettävästä laitteistosta, ei voida luottaa siihen, että ruudunpäivitykseen menisi jokaisessa laitteistossa aina yhtä pitkä aika. Ruudunpäivityksen suurimmalle nopeudelle voidaan asettaa rajoittimia, mutta alarajaan ei voida vaikuttaa. Ratkaisussani käytän tietoa yhden ruudun piirtämiseen kuluva ajasta. Jos objektin halutaan kääntyvän 90 astetta johonkin suuntaan yhdessä sekunnissa, täytyy objektia kääntää yhden piirto-kerran aikana $90 * \text{ruudun piirtämiseen kuluva aika}$. Tällä tavalla menettelemällä sain esimerkkieni objektit liikkumisen käyttäytymään samalla tavalla lait-

teistoltaan eritasoisissa tietokoneissa. Vastaavanlaisiin ongelmiin törmää OpenGL:ssä varmasti usein.

7.2.3. Java 3D

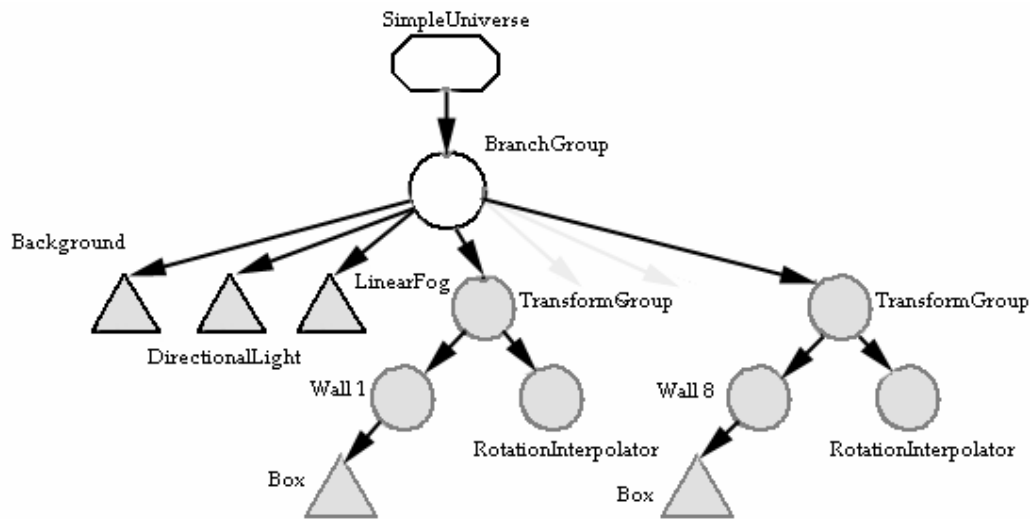
Ennen sovellusten ohjelmoinnin aloittamista täytyi tietokoneeseen ensimmäiseksi asentaa tarvittavat Java-ohjelmointiympäristön osat, eli kääntäjä ja Java 3D API. Molemmat osat ovat saatavissa ilmaiseksi Sunin ylläpitämiltä Java-ohjelmointikielen virallisilta verkkosivuilta (<http://java.sun.com>). Ensimmäiseksi tietokoneeseen kannattaa asentaa Java SDK, jonka uusin versio on tätä kirjoitettaessa 1.4. Tämän jälkeen voidaan siirtyä Java 3D API:n asentamiseen. Tällä hetkellä uusin virallinen versio rajapinnasta on 1.2.1_04, mutta versiosta 1.3 on saatavilla beeta-versio. Tässä tutkielmassa tehdyt sovellukset on toteutettu rajapinnan uusimmalla virallisella versiolla. Java 3D API kannattaa asentaa samaan hakemistoon kuin Java SDK, sillä näin välttyään Java-kääntäjän ja -tulkin tarvitsemien luokkapolkujen (engl. classpath) muuttamiselta.

Ohjelmakoodin kirjoittamiseen on valittavissa useita eri vaihtoehtoja. Javalle on tehty useita graafisia kehitysympäristöjä, joilla voi hallita monipuolisesti projektiaan. Hyviä kokonaan tai osittain ilmaisia kehitysympäristöjä ovat Windows-ympäristössä JCreator (<http://www.jcreator.com>) ja Borlandin JBuilder (<http://www.borland.com/jbuilder>). Ongelmana näissä ja muissa kehitysympäristöissä on se, että niissä ei ole välttämättä sisäänrakennettuna tukea Java 3D API:lle. Tällöin parhaaksi vaihtoehdoksi jää sellaisten tekstieditoreiden käyttäminen, joissa on ohjelmoinnin kannalta hyödyllisiä toimintoja kuten esimerkiksi ohjelmakoodin värikoodaus. Hyviä tekstieditoreita Java-ohjelmointiin ovat Windows-ympäristössä UltraEdit-32 (<http://www.ultraedit.com>) Emacs JDE (<http://jdee.sunsite.dk/>) sekä JPad ja SidePad (<http://www.modelworks.com>).

Sovellusten tekemiseen meni huomattavasti enemmän aikaa, kuin VRML:llä, mutta ohjelmointi oli nopeampaa kuin OpenGL-ohjelmointi. Lisäksi sovelluksille suunnittelin mahdollisimman yksinkertaisen ja toimivan rakenteen, jota pystyisin käyttämään kaikissa kolmessa sovelluksessa. Sovelluksia käytin *com.sun.j3d.utils* -pakkausta, josta käytin maisemagraafin juurena *SimpleUniverse*-luokkaa (ks. kuva 7.1). Juuren lapsena toimii *BranchGroup*-solmu, jonka lapsisolmuina ovat malliin liittyvät valaistusominaisuudet, sumuefekti ja piirrettävät objektit. *TransformGroup*-solmujen lapsina olevien objektien liikkeitä kontrolloivat *RotationInterpolator*-solmut.

Halutusti toimivien sovellusten toteuttaminen oli vaikeampaa kuin VRML:llä, sillä mallin puurakennetta oli aluksi vaikeaa hahmottaa. Varsinkin affiinimuunnosten toteuttaminen oli aluksi uskomattoman hankalaa ja tark-

kuutta vaativaa. Usein objektit eivät käyttäytyneet lainkaan niin kuten niiden toivoi käyttäytyvän. Tähän ongelmaan auttoi mallien puurakenteiden suunnittelu ja pohtiminen paperilla, jonka jälkeen ongelmakohdat löytyivät yleensä nopeasti. Sovelluksissa liikkumiseen käytin esimerkkien mukana tullutta VirtualInput-komponenttia, joka rajoittuneisuudestaan huolimatta on riittävä toteuttamissani sovelluksissa liikkumiseen.



Kuva 7.1. TunnelDemo-esimerkin maisemagraafi

7.3. Piirtonopeuden testaaminen

7.3.1. Johdanto

Esimerkkisovelluksien avulla toteutettiin piirtonopeutta (fps) mittaava testi. Testit ajettiin Windows-käyttöjärjestelmää käyttävissä tietokoneissa, ja testit suoritettiin OpenGL- ja Java 3D –rajapintaa käyttävissä sovelluksissa. VRML:llä toteutetuille esimerkeille ei ajettu testejä, sillä niiden piirtonopeus riippuu katselemiseen käytettävistä ohjelmista ja grafiikkarajapinnoista, joita katseluohjelmat käyttävät. Lisäksi VRML-katseluohjelmista on vaikeata saada luotettavaa tietoa piirtonopeudesta. OpenGL- ja Java 3D –rajapinnoista tarvittavat tiedot ovat helposti saatavilla.

Testituloksia käytettiin kahden kysymyksen tarkasteluun. Kummalla rajapinnalla saa aikaisiksi nopeammin piirrettävää grafiikkaa, kun piirtämistä ei optimoida piirtoa nopeuttavilla tekniikoilla? Minkälaista laitteistoa pitäisi käyttää, että grafiikan piirtäminen olisi riittävän nopeaa? Jälkimmäinen kysymys on tärkeä, sillä kuvaa on piirrettävä vähintään 20 kertaa sekunnissa, jotta liikkuva grafiikka näyttäisi ihmisilmälle sulavalta. Kuitenkin aina on pyrittävä tätäkin nopeutta suurempaan piirtonopeuteen, sillä liian pieni piirtonopeus yhdistettynä käytettävän näytön matalaan virkistystaajuuteen aiheuttaa

havaittavaa kuvan välkkymistä, joka voi aiheuttaa silmän väsymistä. [Kawalsky, 1993; Vince, 1998]

7.3.2. Testin toteutus

Testi toteutettiin kaikissa testilaitteistoissa (katso liite 2) sekä 800*600 että 1024*768 pisteen resoluutiossa. Näytön virkistystaajuudeksi asetin vanhemmilla tietokoneilla 75 Hz ja uusilla tietokoneilla 100 Hz. Päädyin käyttämään eri virkistystaajuutta, koska vanhemmat koneet ja niiden näytöt sekä näyttöohjaimet eivät kyenneet 100 Hz -tilaan. Uudempien koneiden testaaminen 75 Hz -tilassa olisi aiheuttanut piirtonopeuden jäämisen noin 75 fps:n tasolle, ja etukäteen oli jo tiedossa, että ne pystyvät piirtämään esimerkkejä tätä huomattavasti nopeammin.

Toteutin sekä OpenGL- ja Java 3D -esimerkkeihin komponentit, jotka vastaavat piirtonopeuden laskemisesta ja sen tulostamisesta. Java 3D -sovelluksissa piirtonopeuden laskemisesta huolehtii FPSCanvas3D, joka on *Canvas3D*:n aliluokka. *Canvas3D*-luokassa on aliohjelma *postSwap()*, jota kutsutaan aina kun malli on piirretty näytölle. Tähän aliohjelmaan on toteutettu piirtonopeuden laskemiseen tarvittava ohjelmakoodi. OpenGL-sovellukseen tein piirtonopeuden laskemiseen FPSCounter-luokan, jota kutsutaan mallin piirtämiseen käytetystä aliohjelmasta. Molemmissa komponenteissa lasketaan ruudun piirtämiseen kuluva aika piirtämisen aloitus- ja lopetusajan erotuksena. Mittaustulosten parantamiseksi fps-arvo lasketaan viiden ruudun piirtämiseen kuluva ajasta. Kaikissa testisovelluksissa piirtonopeuden tulostamiseen käytettiin FPSLogger-luokkaa. Sen tehtävänä on tulostaa sekunnin välein piirtonopeuden arvo, jonka voi ohjata tulostumaan joko konsoliin tai tekstitiedostoon.

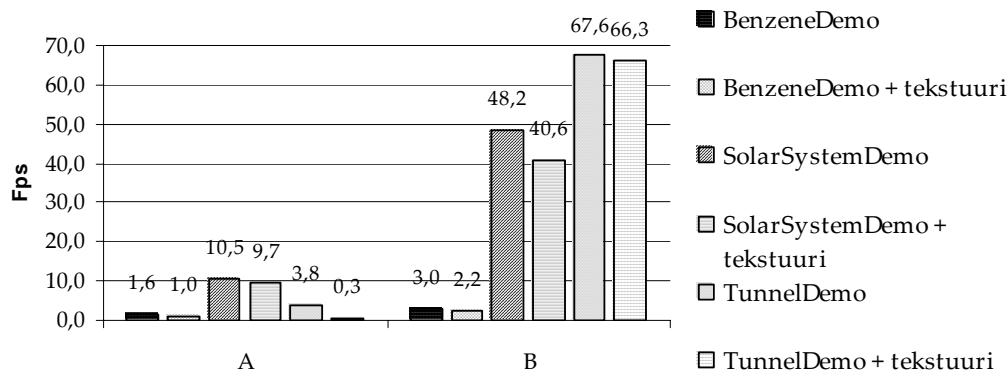
7.3.3. Rajapintojen välinen nopeusero

Testeissä ilmeni, että Java 3D -esimerkkien piirtäminen tapahtui nopeammin kuin OpenGL-esimerkkien. Liitteessä 1 on kerättynä eri laitekoonpanoilla saadut mittaustulokset. Erot olivat selviä varsinkin BenzeneDemoa testattaessa, jossa piirrettävänä useita pallo-objekteja. Pallojen piirtämiseen käytettiin Java 3D -versiossa *com.sun.j3d.utils.geometry* -pakkauksen *Sphere*-luokkaa, ja OpenGL-versiossa *gluSphere()*-aliohjelmaa. On syytä olettaa, että *Sphere*-luokan toteutusta on jollakin tavalla optimoitu, sillä muuten näin suurien erojen syntyminen ei ole mahdollista versioiden välillä. OpenGL-versiossa BenzeneDemon piirtonopeuteen vaikutti myös suuresti pallojen piirtämiseen käytettyjen pintakappaleiden lukumäärä. Java 3D:ssä ei vastaavaa ominaisuutta voinut määritellä, vaan oli silmämääräisesti arvioitava milloin pallot ovat

molemmissa malleissa yhtä tarkat ja samannäköiset. Lisää pallojen vaikutuksesta piirtonopeuteen käsitellään luvussa 7.3.4.

Vaikka Java 3D suoriutuikin piirtämisestä nopeammin, on vaarallista nostaa sitä OpenGL:n edelle. On muistettava, että OpenGL-esimerkit eivät olleet optimoituja, ja Java 3D piirtää optimoitua grafiikkaa. [Java 3D, 2000] TunnelDemo-esimerkissä tunnelin jokainen seinä koostuu kuudesta monikulmiosta, ja niiden piirtämisestä OpenGL suoriutui nopeammin kuin Java 3D. Tämä ero oli selvimmin havaittavissa testissä käytetyillä kahdella hitaammalla koneella. Tehokkailla koneilla ei päässyt syntymään vastaavaa eroa. Erot suuremmalla ja pienemmällä näytöntarkkuudella piirrettäessä olivat uusilla koneilla vähäisiä. Ne suoriutuivat piirtämisestä molemmissa tarkkuustiloissa yhtä nopeasti, koska molemmissa koneissa oli nykyaikaiset OpenGL-yhteensopivat näytönohjaimet. Suurempi resoluutio muodostui liian raskaaksi testin hitaammalle laitteistolle, jossa ei ollut OpenGL-ominaisuuksia tukevaa näytönohjainta. Erot olivat suuria samaa nopeusluokkaa olevan koneen kanssa, jossa oli 3D-kiihdytystä tukeva näytönohjin. Kuvassa 7.2 on havainnollistettu kyseisten laitteistojen piirtonopeuksien eroja, kun ajettiin eri esimerkkien OpenGL-versioita näytöntarkkuuden ollessa 1024*768.

Toteutetut esimerkit pyörivät laitteistoa A (katso liite 2) lukuun ottamatta riittävän nopeasti, että ihmissilmä näki liikkeen tasaisena. Laitteisto B joutui tosin taipumaan BenzeneDemon piirtämisen raskaudelle, ja ruudunpäivitysnopeus jäi sillä liian matalaksi. Uusimmilla laitteilla jokainen esimerkki pyöri riittävän nopeasti. Tämä testi osoittaa, että yksinkertaisista kappaleista koostuvien mallien pyörittämiseen ei välttämättä tarvitse uusinta mahdollista laitekokonpanoa. Tosin ominaisuuksiltaan riittävä 3D-näytönohjin on välttämättömyys.



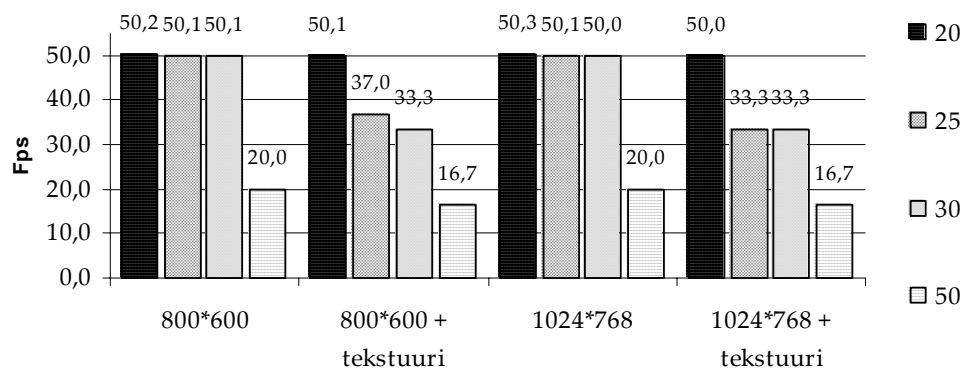
Kuva 7.2. 3D-näytönohjaimen vaikutus piirtonopeuteen

7.3.4. Monikulmioiden lisäämisen vaikutus piirtonopeuteen

BenzeneDemo-esimerkin testeissä törmäsin yllättävän suureen eroon Java 3D - ja OpenGL-versioiden piirtonopeuksien välillä. Tämän eroavaisuuden aiheuttajan löysin piirrettävien pallojen ominaisuuksista. Java 3D -esimerkissä käytin piirtämiseen `com.sun.j3d.utils.geometry` -pakkauksen `Sphere`-luokkaa, jonka piirtämän pallon piirto-ominaisuuksiin ei voinut vaikuttaa. OpenGL-esimerkissä pallojen piirtäminen tehtiin GLU-kirjaston `gluSphere()`-aliohjelmalla.

Aliohjelmalle on määriteltävä pallon säteen lisäksi myös kuinka monesta osasta pallo koostuu leveys- ja pituussuunnassa. Mitä suurempi arvo on, sitä enemmän monikulmioita pallon piirtämiseen käytetään ja sitä pyöreämmältä se näyttää. Monikulmioiden määrän kasvattaminen vaikuttaa myös hidastavasti mallin piirtonopeuteen. Koska tietoa Java 3D:n käyttämän pallon ominaisuuksista ei löytynyt, valitsin sopivan arvon silmämääräisesti OpenGL-esimerkkiin. Sopivaksi arvoksi valikoitui 20. Kuitenkin piirtonopeus oli huomattavasti hitaampi kuin Java 3D -versiossa. Pudottamalla arvon kymmeneen päästiin suunnille yhtä nopeaan piirtonopeuteen. Kuitenkin pallot muuttuivat tällöin liian kulmikkaiksi.

Pallojen käyttäminen virtuaalimaailmoissa voi aiheuttaa huomattavaa piirtonopeuden hidastumista, varsinkin jos palloista halutaan tarkkoja ja hyvännäköisiä. Tätä piirtonopeuden hidastumista havainnollistetaan kuvassa 7.3, jonka arvot on saatu BenzeneDemon OpenGL-versiosta, jota ajettiin laitteistolla C, kun näytöntarkkuus oli 1024*768. Jos mallissaan halua käyttää palloja tai muita kaarevia pintoja, on tällöin syytä perehtyä piirtämistä nopeutaviin tekniikoihin. Tällöin kysymykseen tulee esimerkiksi kamerasta kaukana olevien pallo-objektien yksityiskohtien karsiminen ja kamerassa näkymättömien objektien piirtämättä jättäminen. [Kokkarinen *et al.*, 2001]



Kuva 7.3. Pallon osien lukumäärän vaikutus piirtonopeuteen

8. Lopuksi

8.1. Tulevaisuudennäkymät

Tutkielmassa käsiteltyjen toteutusvaihtoehtojen tulevaisuudennäkymät ovat erilaisia. OpenGL on jo saavuttanut vankan sijan kolmiulotteisen grafiikan esittämiseen käytettyjen rajapintojen joukossa. Sen etuna on laiteriippumattomuus, joka takaa sen, että OpenGL-rajapintaa käytetään myös tulevaisuuden projekteissa. Linux-ympäristössä se on ainoa vaihtoehto matalantason kolmiulotteisen grafiikan esittämiseen. Windows-ympäristössä OpenGL kilpailee tiukasti Microsoftin Direct3D-rajapinnan kanssa. Tämä voi aiheuttaa ongelmia OpenGL:lle tulevaisuudessa, sillä Microsoftin tuki omalle rajapinnalleen on laajempaa. OpenGL-rajapinnan kehitystä kontrolloi yhteistyöelin (OpenGL ARB), joka päättää rajapinnan kehityssuunnista. Koska yhteistyöelin koostuu useista tietotekniikan suurista laite- ja ohjelmistovalmistajista, on syytä odottaa OpenGL-rajapinnan kehittyvän myös tulevaisuudessa. Tällä hetkellä on olemassa suunnitelmia OpenGL 2.0:n sisältämistä ominaisuuksista.

Java 3D -rajapinnan tulevaisuus on kysymysmerkki, sillä Sun on vähentänyt rajapinnan kehitykseen käytettävää taloudellista panostaan [J3D.org, 2002]. Java 3D:n nykytilanne vastaa hyvin pitkälle Java-ohjelmointikielen alkutilannetta. Sekin oli täynnä lupauksia, mutta vasta verkkokäyttö nosti sen menestykseen. Silti todelliset ”tappajasovellukset” puuttuvat edelleen. Lisäksi Java-ohjelmointikielen riesana ovat myös yleiset ennakkoluulot sen hitaudesta. Java 3D -ohjelmien pääasiallinen käyttökohde voisi tulevaisuudessa hyvinkin olla erilaiset verkkosovellukset.

VRML loi ilmestyessään suuren buumin. Sen ympärille syntyi lukuisia katselu- ja mallinnusohjelmia sekä tuottajia että konsultteja. VRML:lle ei kuitenkaan löytynyt liiketaloudellista käyttöä, ja buumi loppui lyhyeen. VRML:n korvaaja voi olla Extensible 3D (X3D). X3D-kieli pitäisi olla edeltäjänsä kevyempi ja vuorovaikutteisempi, mutta nämä ominaisuudet eivät vielä yksin takaa kielen menestymistä. Yksi VRML:n suurimpia kompastuskiviä oli se, että mallien katselemiseen tarvitsi erillisen katseluohjelman. Kaikki käyttäjät eivät viitsi tai osaa hankkia katselemiseen tarvittavaa ohjelmistoa. X3D-kielen menestyminen riippuu hyvin paljon myös ohjelmistovalmistajista. Jos ne tarjoavat mahdollisuuden X3D-sovellusten katselemisen esimerkiksi osana www-selaimiaan, niin X3D-sovellusten käyttö yleistyy. Sama myös pätee myös Java 3D:n menestymiseen. Helppo ja yksinkertainen käyttöönotto toisi lisää sovellusten tekijöitä ja mahdollisesti myös asiakkaita. Tällöin myös markkinat kiinnostuisivat tekniikasta.

8.2. Loppusanat

Tutkielmassa tutustuttiin virtuaalimaailmojen toteuttamiseen soveltuviin vaihtoehtoihin (VRML, OpenGL ja Java 3D), sekä toteutettiin jokaisella vaihtoehdolla kolme esimerkkisovellusta. Lisäksi esimerkeille toteutettiin testit, joiden avulla mitattiin OpenGL- ja Java 3D -sovellusten piirtonopeuksia. Kokeiden avulla selvisi, että optimoimattomissa malleissa ei piirtonopeuksien välillä esiintynyt merkittävää eroa. Yksinkertaisista monikulmioista koostuvaa OpenGL-mallia piirrettiin hieman nopeammin kuin vastaavaa Java 3D -mallia.

Tutkittava aihe on ollut erittäin mielenkiintoinen. Aihealue on melkoisen laaja, joten käsiteltyjen toteutusvaihtoehtojen eri ominaisuuksien yksityiskohtainen tutkimus on jäänyt vähäiseksi. Esimerkiksi piirtämiseen liittyviä optimointimenetelmiä olisi ollut mielenkiintoista tutkia tarkemmin. Niiden käytämisen tarpeellisuus tuli huomattua toteutetuissa testeissä, joilla tehokkaampikin laitteisto saatiin kummasti tukehtumaan. Henkilökohtaisesti aihe tarjosi minulle runsaasti oivaltamisen hetkiä ja uusia taitoja, sillä käsitellyt aiheet olivat minulle Java- ja C++-ohjelmointikieliä lukuun ottamatta uusia.

Virtuaalimaailmojen toteuttamisen tutkimista kannattaisi jatkaa pienemmissä osissa. Yhdeksi mahdolliseksi jatkotutkimusaiheeksi voisi rajata tässä tutkielmassa sivuttujen muiden ohjelmointikielten tai -rajapintojen ominaisuuksien tutkimisen. Tutkimisen arvoinen toteutuskieli voisi olla esimerkiksi Extensible 3D (X3D) ja rajapinta DirectX. Toinen hyvä jatkotutkimuksen aihe olisi kolmiulotteiseen piirtämiseen liittyvät tekniikat ja algoritmit, joilla voidaan nopeuttaa mallin piirtämistä. Näitä ovat muun muassa piilopintojen poisto ja piirrettävien yksityiskohtien karsiminen. Muutaman vuoden kuluttua olisi jälleen mielenkiintoista kartoittaa yleisesti virtuaalimaailman toteuttamiseen soveltuvia vaihtoehtoja ja niiden tarjoamia ominaisuuksia, koska uusia vaihtoehtoja syntyy jatkuvasti tekniikan kehittyessä.

Viiteluettelo

- [3D Dictionary] 3D Computer Graphics Dictionary. Saatavilla osoitteesta: <http://oss.medialab.chalmers.se/dictionary/>. Viitattu 17.6.2002.
- [AlphaWorld, 2001] AlphaWorld, 2001. Katso: <http://www.activeworlds.com/worlds/alphaworld/>. Viitattu 17.6.2002.
- [AlphaWorld Map] About the AlphaWorld Map. Saatavilla osoitteesta: <http://mapper.activeworlds.com/aw/about-full.html>. Viitattu 17.6.2002.
- [ArtMuseum.net, 2000] Sensora, 2000. Saatavilla osoitteesta: <http://www.artmuseum.net/w2vr/archives/Fisher/Environment.html#Sensorama>. Viitattu 17.6.2002.
- [ATK-Sanakirja, 1999] Tietotekniikan liitto ry:n sanastotoimikunta, *Tietotekniikan liiton atk-sanakirja 1999*. Suomen Atk-kustannus Oy, 1999.
- [Background paper, 1995] Distributed interactive simulation of combat, 1995. Saatavilla osoitteesta: http://www.wws.princeton.edu/~ota/disk1/1995/9512_n.html. Viitattu 17.6.2002.
- [Bank, 1995] David Bank, The Java Saga, 1995. Saatavilla osoitteesta: http://www.wired.com/wired/archive/3.12/java.saga_pr.html. Viitattu 17.6.2002.
- [Bouvier, 2001] Dennis J Bouvier, Getting started with the Java 3D API, 2001. Saatavilla osoitteesta: <http://java.sun.com/products/java-media/3D/collateral/>. Viitattu 17.6.2002.
- [Brits, 2001] Lionel Brits, Matrices: An Introduction, 2001. Saatavilla osoitteesta: <http://www.cadvision.com/britsc/tgc/tutorials/matrices.htm>. Viitattu 17.6.2002.
- [Cant *et al.*, 2001] Richard Cant, Nathan Chia, David Al-Dabass, New anti-aliasing and depth of field techniques for games. The Nottingham Trent University, Dept. of Computing and Mathematics, 2001. Saatavilla osoitteesta: <http://ducati.doc.ntu.ac.uk/uksim/dad/webpagepapers/Game-18.pdf>. Viitattu 17.6.2002.
- [Carey *et al.*, 1997] Rikk Carey, Gavin Bell, Chris Marrin, ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97), 1997. Saatavilla osoitteesta: <http://www.web3d.org/technicalinfo/specifications/vrml97/vrml97specification.pdf>. Viitattu 17.6.2002.
- [Damer, 1996] Bruce Damer, Inhabited virtual worlds: a new frontier for interaction design. *ACM* **3**, 5 (Sept./Oct. 1996), 27-34.
- [de Oliveira *et al.*, 2000] Jauvane C. De Oliveira, Shervin Shirmohammadi and Nicolas D. Georganas, A Collaborative Virtual Environment for Industrial Training. In: *Proceedings IEEE Virtual Reality 2000 Conference*, 288.

- [Direct3D, 2002] Introduction to DirectX Graphics, 2002. Saatavilla osoitteesta: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dx8_c/hh/dx8_c/graphics_intro_9gvn.asp. Viitattu 17.6.2002.
- [Elias, 2001] Hugo Elias, The good-looking textured light-sourced bouncy fun smart and stretchy page. Saatavilla osoitteesta: <http://freespace.virgin.net/hugo.elias/>. Viitattu 17.6.2002.
- [GLUT] GLUT – OpenGL Utility Toolkit, saatavilla osoitteesta: <http://www.opengl.org/developers/documentation/glut/index.html>. Viitattu 17.6.2002.
- [Hintikka *et al.*, 1998] Kari A. Hintikka, Ilpo Kojo, Markku Metsämäki, *Virtuaaliympäristöjen suunnitteluopas*. Oy Edita Ab, 1998.
- [Hosseini and Georganas, 2001] Mojtaba Hosseini and Nicolas D. Georganas, Collaborative Virtual Environments for Training, 2001. Saatavilla osoitteesta: <http://www.acm.org/sigmm/mm2001/ep/hosseini-demo>. Viitattu 17.6.2002.
- [Java 3D, 2000] The Java 3D™ API Specification Version 1.2, 2000. Saatavilla osoitteesta: http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dguide/index.html. Viitattu 17.6.2002.
- [Java 3D API, 2000] The Java 3D™ API Documentation Version 1.2, 2000. Saatavilla osoitteesta: http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dapi/index.html. Viitattu 17.6.2002.
- [Java 3D Releases, 2002] Previous Releases of Java 3D™ API, 2002. Saatavilla osoitteesta: <http://java.sun.com/products/java-media/3D/olderreleases.html>. Viitattu 17.6.2002.
- [Java Timeline, 2000] The Java™ Platform: Five Years in Review, 2000. Saatavilla osoitteesta: <http://java.sun.com/features/2000/06/time-line.html>. Viitattu 17.6.2002.
- [J3D.org, 2002] The Java 3D Community Site, 2002. Katso: <http://www.j3d.org>. Viitattu 17.6.2002.
- [John and Riding, 1999] Nigel W. John and Mark Riding, Surgical Simulators on the World Wide Web – This Must Be The Way Forward? Manchester Visualization Centre, University of Manchester, 1999. Saatavilla osoitteesta: <http://synaptic.mvc.mcc.ac.uk/Papers/ukvrsig99.pdf>. Viitattu 17.6.2002.
- [Järvinen, 1999] Aki Järvinen, Digitaaliset pelit ja pelikulttuurit. Kirjassa: Aki Järvinen ja Ilkka Mäyrä (toim.), *Johdatus Digitaaliseen Kulttuuriin*. Vastapaino, 1999.
- [Kalawsky, 1993] Roy S. Kalawsky, *The Science of Virtual Reality and Virtual Environments*. Addison-Wesley, 1993.

- [Kilgard, 1996] Mark J. Kilgard, The OpenGL Utility Toolkit (GLUT) Programming Interface – API Version 3, 1996. Saatavilla osoitteesta <http://www.opengl.org/developers/documentation/glut/glut-3.spec.pdf>. Viitattu 17.6.2002.
- [Kokkarinen *et al.*, 2001] Ilkka Kokkarinen, Wille Kuutti, Juha Nieminen, *Tietokonegrafiikka*. Satku, 2001.
- [Koskimies, 2000] Kai Koskimies, *Oliokirja*. Satku, 2000.
- [Kurhila, 1997] Jaakko Kurhila, Tietokone GOA – grafiikan oppiaineisto, 1997. Saatavilla osoitteesta: <http://www.cs.helsinki.fi/group/goa/index.html>. Viitattu 17.6.2002.
- [Linturi *et al.*, 2000] Risto Linturi, Marja-Riitta Koivunen, Jari Sulkanen, Helsinki Areena 2000 – Augmenting a Real City to a Virtual One. In: Toru Ishida, Katherine Isbister (eds.), *Digital Cities - Technologies, Experiences, and Future Perspectives*. Springer Verlag, 2000, 83-96.
- [Morningstar and Farmer, 1990] Chip Morningstar and F. Randall Farmer, The Lessons of Lucasfilm's Habitat, 1990. Saatavilla osoitteesta: <http://scara.com/~ole/literatur/LessonsOfHabitat.html>. Viitattu 17.6.2002.
- [Neider *et al.*, 1994] Jackie Neider, Tom Davis, Mason Woo, OpenGL Programming Guide – The Official Guide to Learning OpenGL, Release 1, 1994. Saatavilla osoitteesta: <http://fly.cc.fer.hr/~unreal/theredbook/>. Viitattu 17.6.2002.
- [OpenFlight, 2002] OpenFlight API, 2002. Katso: <http://www.multigen.com/products/standards/openflight/index.shtml>. Viitattu 17.6.2002.
- [OpenGL, 2002] OpenGL, 2002. Katso: <http://www.opengl.org>. Viitattu 17.6.2002.
- [OpenGL ARB, 2002] OpenGL Architectural Review Board, 2002. Katso: <http://www.opengl.org/developers/about/arb.html>. Viitattu 17.6.2002.
- [OpenGL Performer, 2002] OpenGL Performer, 2002. Katso: <http://www.sgi.com/software/performer/>. Viitattu 17.6.2002.
- [OpenGL Tutorial, 1997] OpenGL Tutorial, 1997. Saatavilla osoitteesta: <http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduction.html>. Viitattu 17.6.2002.
- [Open Inventor, 2002] Open Inventor, 2002. Katso: <http://www.sgi.com/software/inventor/>. Viitattu 17.6.2002.
- [OpenGVS, 2002] OpenGVS, 2002. Katso: <http://www.opengvs.com/>. Viitattu 17.6.2002.
- [SDL] Simple DirectMedia Layer. Katso: <http://www.libsdl.org>. Viitattu 17.6.2002.
- [Segal and Akeley, 1997] Mark Segal, Kurt Akeley, The Design of the OpenGL Graphics Interface, 1997. Saatavilla osoitteesta: http://www.opengl.org/developers/documentation/white_papers/opengl/opengl.html. Viitattu 17.6.2002.

- [SGI, 2002] Silicon Graphics Inc. Katso: <http://www.sgi.com>.
- [Sohl] Barry A. Sohl, Distributed Simulation Concepts and Applications, saatavilla osoitteesta: <http://www.cecs.csulb.edu/~sohl/distributedsim/milapps/milapps-body.html>. Viitattu 17.6.2002.
- [Slater *et al.*, 2001] Mel Slater, Anthony Steed and Yiorgos Chrysanthou, *Computer Graphics and Virtual Enviroments From Realism to Real-time*. Addison-Wesley, 2001. Viitattu 17.6.2002.
- [Smith *et al.*, 1996] David Smith, Richard Boyd, Alan Scott, *VRML & 3D*. Hayden Books. Suomenkielisen version on julkaissut Suomen Atk-kustannus Oy, 1996.
- [Sun OpenGL] Sun OpenGL Man Pages, saatavilla osoitteesta: <http://www.sun.com/software/graphics/OpenGL/manpages/man3glindex.html>. Viitattu 17.6.2002.
- [Vapourtech, 2001] Floppy's VRML97 Tutorial, 2001. Saatavilla osoitteesta: <http://web3d.vapourtech.com/tutorials/vrml97/>. Viitattu 17.6.2002.
- [Vince, 1995] John Vince, *Virtual Reality Systems*. Addison-Wesley, 1995.
- [VRML, 1998] History of VRML Specification, 1998. Saatavilla osoitteesta: <http://www.web3d.org/about/historyofvrml.html>. Viitattu 17.6.2002.
- [VRML, 2002] VRML, 2002. Katso: <http://www.vrml.org>. Viitattu 17.6.2002.
- [Woo *et al.*, 1999] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, OpenGL Architecture Review Board, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.
- [X3D, 2002] X3D, 2002. Katso: http://www.web3d.org/fs_x3d.htm. Viitattu 17.6.2002.
- [Äänipää, 2001] Äänipää, 2001. Katso: <http://www.cult.tpu.fi/sound/>. Viitattu 17.6.2002.

Testitulokset

Java 3D - 800 * 600	A	B	C	D
BenzeneDemo	2,8	5,1	73,0	75,0
BenzeneDemo + tekstuurit	2,0	5,3	75,8	75,1
SolarSystemDemo	17,1	70,9	100,0	100,0
SolarSystemDemo + tekstuurit	15,9	56,4	99,0	100,0
TunnelDemo	4,3	64,6	100,0	100,1
TunnelDemo + tekstuurit	0,8	56,3	100,0	100,0

Java 3D - 1024 * 768	A	B	C	D
BenzeneDemo	2,5	5,6	75,0	74,9
BenzeneDemo + tekstuurit	1,6	4,7	70,3	70,3
SolarSystemDemo	11,6	76,2	100,0	100,1
SolarSystemDemo + tekstuurit	10,8	55,8	99,9	99,9
TunnelDemo	2,7	57,7	100,0	100,0
TunnelDemo + tekstuurit	0,5	55,5	99,9	100,0

OpenGL - 800 * 600	A	B	C	D
BenzeneDemo	2,0	3,3	50,2	50,2
BenzeneDemo + tekstuurit	1,1	2,5	50,1	50,2
SolarSystemDemo	14,5	57,9	100,3	100,3
SolarSystemDemo + tekstuurit	13,4	48,1	100,1	100,2
TunnelDemo	6,5	75,0	100,1	100,2
TunnelDemo + tekstuurit	0,4	75,0	100,1	100,2

OpenGL - 1024 * 768	A	B	C	D
BenzeneDemo	1,6	3,0	50,3	50,4
BenzeneDemo + tekstuurit	1,0	2,2	50,0	50,2
SolarSystemDemo	10,5	48,2	99,9	100,4
SolarSystemDemo + tekstuurit	9,7	40,6	100,1	100,2
TunnelDemo	3,8	67,6	100,0	100,1
TunnelDemo + tekstuurit	0,3	66,3	100,0	100,0

Testeissä käytetyt laitekoonpanot

Laitekoonpanot	A	B	C	D
Prosessori	Intel Pentium II	Intel Pentium II	AMD Athlon	AMD Athlon XP 1600+
Kellotaajuus	266MHz	300MHz	1.33GHz	1.40GHz
Muisti	64 MB	350 MB	256 MB	256 MB
Käyttöjärjestelmä	Windows 98	Windows 2000	Windows 2000	Windows XP
Näytönohjain	Intel740	Matrox Millennium G400 DualHead	NVIDIA GeForce2 Pro	NVIDIA GeForce3 Ti 200
Näytönohjaimen muisti	8 MB	16 MB	64 MB	64 MB
Näytön värin määrä	True Color	True Color	True Color	True Color
Näytön virkistystaajuus	75 Hz	75 Hz	100 Hz	100 Hz
OpenGL-kihdit	Ei	Kyllä	Kyllä	Kyllä

TunnelDemo.wrl

```

#VRML V2.0 utf8

Background {
  skyColor 0.0 0.0 0.0
}

Fog {
  color 1 1 1
  fogType "LINEAR"
  visibilityRange 0
}

DEF World Transform {
  rotation 0.0 0.0 0.0 0.0
  children [
    DirectionalLight {
      color 1.0 1.0 1.0
      on TRUE
      direction -1.0 -1.0 -1.0
      intensity 1.0
      ambientIntensity 1.0
    }
    DEF Tunnel Transform {
      children [
        #Seinä 1
        Transform {
          translation 0.0 6.05 0.0
          rotation 0.0 0.0 0.0 0.0

          children DEF Wall1
          Shape {
            appearance Appearance {
              material Material {
                diffuseColor 0.5 0.5 0.5
                specularColor 1.0 1.0 1.0
                emissiveColor 0.5 0 0.5
              }
              texture ImageTexture {
                url "ttimage.jpg"
              }
            }
            geometry Box {
              size 5.0 0.1 80.0
            }
          }
        },
        #Seinä 2
        Transform {
          translation 4.28 4.28 0.0
          rotation 0.0 0.0 1.0 -0.785
          children DEF Wall2
          Shape {
            appearance Appearance {
              material Material {
                diffuseColor 0.0 0.0 0.5
                diffuseColor 0.5 0.5 0.5
                specularColor 1.0 1.0 1.0
                emissiveColor 0.0 0.0 0.5
              }
              texture ImageTexture {
                url "ttimage.jpg"
              }
            }
            geometry Box {
              size 5.0 0.1 80.0
            }
          }
        },
        #Seinä 3
        Transform {
          translation 6.05 0.0 0.0
          rotation 0.0 0.0 1.0 1.571
          children USE Wall1
        },
        #Seinä 4
        Transform {
          translation 4.28 -4.28 0.0
          rotation 0.0 0.0 1.0 0.785
          children USE Wall2
        }
      ]
    }
  ]
}

```

```

    },
    #Seinä 5
    Transform {
        translation 0.0 -6.05 0.0
        rotation 0.0 0.0 0.0 0.0
        children USE Wall1
    },
    #Seinä 6
    Transform {
        translation -4.28 -4.28 0.0
        rotation 0.0 0.0 1.0 -0.785
        children USE Wall2
    },
    #Seinä 7
    Transform {
        translation -6.05 0.0 0.0
        rotation 0.0 0.0 1.0 1.571
        children USE Wall1
    },
    #Seinä 8
    Transform {
        translation -4.28 4.28 0.0
        rotation 0.0 0.0 1.0 0.785
        children USE Wall2
    }
}
]
}
DEF Clock TimeSensor {
    loop TRUE
    cycleInterval 4.0
}
DEF TunnelRotation OrientationInterpolator {
    key [ 0.0 0.5 1.0]
    keyValue [
        0.0 0.0 1.0 0.0
        0.0 0.0 1.0 3.14
        0.0 0.0 1.0 6.28
    ]
}
]
}
}

ROUTE Clock.fraction_changed TO TunnelRotation.set_fraction
ROUTE TunnelRotation.value_changed TO Tunnel.set_rotation

```

TunnelDemo.java

```

import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.image.TextureLoader;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

class Wall
{
    private final String WALL_TEXTURE = "../images/ttimage.jpg";
    private Applet m_td;
    private boolean m_texture;

    public Wall( Applet ss, boolean texture )
    {
        m_td = ss;
        m_texture = texture;
    }

    public TransformGroup createWall( final float xpos, final float ypos,
        final float zpos, final float radians, final Color3f wColor )
    {
        Color3f white = new Color3f( 1.0f, 1.0f, 1.0f );
        Color3f black = new Color3f( 0.0f, 0.0f, 0.0f );
        Color3f diffuse = new Color3f( 0.5f, 0.5f, 0.5f );

        TextureLoader tex = null;
        TextureAttributes texAttr = null;

        //Luodaan pintamateriaalin ominaisuudet
        Appearance a = new Appearance();
        ColoringAttributes ca = new ColoringAttributes( wColor,
            ColoringAttributes.NICEST );
        a.setColoringAttributes( ca );
        a.setMaterial( new Material( black, wColor, diffuse, white, 80.0f ) );

        //Ladataan käytettävä tekstuuri
        tex = new TextureLoader( WALL_TEXTURE, m_td );
        a.setTexture( tex.getTexture() );
        texAttr = new TextureAttributes();
        texAttr.setTextureMode( TextureAttributes.MODULATE );
        a.setTextureAttributes( texAttr );

        Transform3D tr = new Transform3D();
        tr.setTranslation( new Vector3f( xpos, ypos, zpos ) );
        Transform3D rot = new Transform3D();
        rot.rotZ( radians );
        tr.mul( rot );
        TransformGroup wall = new TransformGroup( tr );
        wall.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );
        wall.setCapability( TransformGroup.ALLOW_TRANSFORM_READ );
        wall.addChild( new Box( 5.0f, 0.1f, 80.0f, Box.GENERATE_NORMALS |
            Box.GENERATE_TEXTURE_COORDS, a ) );

        return wall;
    }
}

public class TunnelDemo extends Applet
{
    private SimpleUniverse m_su = null;
    private Canvas3D m_c = null;
    private InputDevice m_device = null;
    private SensorBehavior m_s = null;

    public BranchGroup createSceneGraph()
    {
        BranchGroup objRoot = new BranchGroup();
        BoundingSphere bounds = new BoundingSphere( new Point3d( 0.0,0.0,0.0 ),
            100.0 );

        //Tausta
        Color3f bgColor = new Color3f( 0.05f, 0.05f, 0.2f );
        Background bg = new Background( bgColor );
        bg.setApplicationBounds( bounds );
    }
}

```

```

objRoot.addChild(bg);

//Valot
Color3f lColor1 = new Color3f( 1.0f, 1.0f, 1.0f );
Vector3f lDir1 = new Vector3f( -1.0f, -1.0f, -1.0f );

DirectionalLight lgt1 = new DirectionalLight( lColor1, lDir1 );
lgt1.setInfluencingBounds( bounds );
objRoot.addChild( lgt1 );

//Sumu
LinearFog fog = new LinearFog();
fog.setColor( new Color3f( 0.5f, 0.5f, 0.5f ) );
fog.setFrontDistance( 55.0d );
fog.setBackDistance( 200.0d );
fog.setCapability( Fog.ALLOW_COLOR_WRITE );
fog.setCapability( LinearFog.ALLOW_DISTANCE_WRITE );
fog.setInfluencingBounds( bounds );
objRoot.addChild( fog );

//Objektit
objRoot.addChild( createObject( 0.0f, 6.05f, 0.0f, 0.0f,
    new Color3f( 0.5f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( 4.28f, 4.28f, 0.0f, -0.785f,
    new Color3f( 0.0f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( 6.05f, 0.0f, 0.0f, 1.571f,
    new Color3f( 0.5f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( 4.28f, -4.28f, -2.0f, 0.785f,
    new Color3f( 0.0f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( 0.0f, -6.05f, 0.0f, 0.0f,
    new Color3f( 0.5f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( -4.28f, -4.28f, 0.0f, -0.785f,
    new Color3f( 0.0f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( -6.05f, -2.0f, 2.0f, 1.571f,
    new Color3f( 0.5f, 0.0f, 0.5f ), false ) );
objRoot.addChild( createObject( -4.28f, 4.28f, 0.0f, 0.785f,
    new Color3f( 0.0f, 0.0f, 0.5f ), false ) );

return objRoot;
}

private Group createObject( final float xpos, final float ypos,
    final float zpos, final float radians, final Color3f wColor,
    final boolean texture )
{
    TransformGroup objTrans = new TransformGroup();
    objTrans.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );
    objTrans.setCapability( TransformGroup.ALLOW_TRANSFORM_READ );
    TransformGroup obj;
    Wall w = new Wall( this, m_texture );
    obj = w.createWall( xpos, ypos, zpos, radians, wColor );
    objTrans.addChild( obj );

    //Pyörimistä kontrolloiva interpolaattori
    Transform3D zAxis = new Transform3D();
    zAxis.rotX( Math.PI/2.0f );
    Alpha rotationAlpha = new Alpha( -1, Alpha.INCREASING_ENABLE, 0, 0,
        4000, 0, 0, 0, 0, 0 );
    RotationInterpolator rotator = new RotationInterpolator( rotationAlpha,
        objTrans, zAxis, 0.0f, (float) Math.PI*2.0f );
    BoundingSphere bounds = new BoundingSphere( new Point3d( 0.0,0.0,0.0 ),
        100.0 );
    rotator.setSchedulingBounds( bounds );
    objTrans.addChild( rotator );
    return objTrans;
}

public TunnelDemo(){}

public void init()
{
    m_device = new VirtualInputDevice();
    setLayout( new BorderLayout() );
    GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();

    m_c = new Canvas3D( config );
    add( "Center", m_c );

    // Luodaan yksinkertainen maisemagraafi
    BranchGroup scene = createSceneGraph();
    m_su = new SimpleUniverse( m_c );
}

```

```
m_device.initialize();
//VirtualInputDevicen liittämien malliin
m_su.getViewer().getPhysicalEnvironment().addInputDevice( m_device );
TransformGroup viewTrans =
    m_su.getViewingPlatform().getViewPlatformTransform();
m_s = new SensorBehavior( viewTrans, m_device.getSensor( 0 ) );
m_s.setSchedulingBounds( new BoundingSphere(
    new Point3d( 0.0, 0.0, 0.0 ), Float.MAX_VALUE ) );

scene.addChild( m_s );
m_su.addBranchGraph( scene );
}
public void destroy()
{
    m_su.removeAllLocales();
}

public static void main( String[] args )
{
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    Dimension dimension = toolkit.getScreenSize();
    new MainFrame( new TunnelDemo(), dimension.width, dimension.height );
}
}
```

TunnelDemo.cpp

```

#include <stdlib.h>
#include <GL/glut.h>
#include <stdio.h>
#include <string.h>
#include "BMPLoader.h"
#include "FPSCounter.h"

static GLfloat theta[] = {0.0, 0.0, 0.0};
static GLfloat viewer[] = {0.0, 0.0, 0.0};
const int m_numTextures = 1;
GLdouble m_rotate = 0.001;
GLuint m_textureName[m_numTextures];
GLfloat m_ambient_color[] = { 0.0, 0.0, 0.0 };
GLfloat m_fog_color[] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat m_emissive_color1[] = { 0.5, 0.0, 0.5 };
GLfloat m_emissive_color2[] = { 0.0, 0.0, 0.5 };
GLfloat m_diffuse_color[] = { 0.5, 0.5, 0.5 };
GLfloat m_specular_color[] = { 1.0, 1.0, 1.0 };
GLfloat m_shininess[] = { 80.0 };
double rotation = 0.0;
#define TUNNEL 1

void make_tunnel()
{
    glGenLists( TUNNEL, GL_COMPILE );
    glMaterialfv( GL_FRONT, GL_AMBIENT, m_ambient_color );
    glMaterialfv( GL_FRONT, GL_DIFFUSE, m_diffuse_color );
    glMaterialfv( GL_FRONT, GL_SPECULAR, m_specular_color );
    glMaterialfv( GL_FRONT, GL_SHININESS, m_shininess );

    // Etummainen tahko
    glNormal3d( 0, 0, 1 ); // Pintaa kohtisuorassa oleva vektori
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 0.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 5.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 5.0, 0.5, 0.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 0.0, 0.5, 0.0 );
    glEnd();
    // Taaimmainen tahko
    glNormal3d( 0, 0, -1 );
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 0.0, 0.0, -80.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 5.0, 0.0, -80.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 5.0, 0.5, -80.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 0.0, 0.5, -80.0 );
    glEnd();
    // Päälimmäinen tahko
    glNormal3d( 0, 1, 0 );
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 0.0, 0.5, 0.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 5.0, 0.5, 0.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 5.0, 0.5, -80.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 0.0, 0.5, -80.0 );
    glEnd();
    // Alimmainen tahko
    glNormal3d( 0, -1, 0 );
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 0.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 5.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 5.0, 0.0, -80.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 0.0, 0.0, -80.0 );
    glEnd();
    // Oikea tahko
    glNormal3d( 1, 0, 0 );
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 5.0, 0.5, 0.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 5.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 5.0, 0.0, -80.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 5.0, 0.5, -80.0 );
    glEnd();
    // Vasen tahko
    glNormal3d( -1, 0, 0 );
    glBegin(GL_QUADS);
    glTexCoord2d( 0.0, 0.0 ); glVertex3d( 0.0, 0.5, 0.0 );
    glTexCoord2d( 1.0, 0.0 ); glVertex3d( 0.0, 0.0, 0.0 );
    glTexCoord2d( 1.0, 1.0 ); glVertex3d( 0.0, 0.0, -80.0 );
    glTexCoord2d( 0.0, 1.0 ); glVertex3d( 0.0, 0.5, -80.0 );
    glEnd();
}

```

```

    glEndList();
}

void display()
{
    rotation += ( 1/m_rotate );

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity();
    glTranslatef( viewer[0], viewer[1], viewer[2] );
    glRotatef( theta[0], 1.0, 0.0, 0.0 );
    glRotatef( theta[1], 0.0, 1.0, 0.0 );
    glRotatef( theta[2], 0.0, 0.0, 1.0 );

    glEnable( GL_TEXTURE_2D );
    glBindTexture( GL_TEXTURE_2D, m_textureName[0] );

    gluLookAt( 0.0, 0.0, 0.0,
               0.0, 0.0, -20.0,
               0.0, 1.0, 0.0 );
    //Tunnelin pyöriminen
    glRotated( rotation * 90, 0.0, 0.0, 1.0 );

    //Seinä 1
    glPushMatrix();
    glTranslatef( 2.5, 6.55, 0.0 );
    glRotatef( 180.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color1 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 2
    glPushMatrix();
    glTranslatef( 6.25, 2.85, 0.0 );
    glRotatef( 135.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color2 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 3
    glPushMatrix();
    glTranslatef( 6.0, -2.45, 0.0 );
    glRotatef( 90.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color1 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 4
    glPushMatrix();
    glTranslatef( 2.5, -6.05, 0.0 );
    glRotatef( 45.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color2 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 5
    glPushMatrix();
    glTranslatef( -2.5, -6.05, 0.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color1 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 6
    glPushMatrix();
    glTranslatef( -6.05, -2.5, 0.0 );
    glRotatef( -45.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color2 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 7
    glPushMatrix();
    glTranslatef( -6.15, 2.85, 0.0 );
    glRotatef( -90.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color1 );
    glCallList( TUNNEL );
    glPopMatrix();

    // Seinä 8
    glPushMatrix();
    glTranslatef( -2.75, 6.75, 0.0 );
    glRotatef( 225.0, 0.0, 0.0, 1.0 );
    glMaterialfv( GL_FRONT, GL_EMISSION, m_emissive_color2 );

```

```

    glCallList( TUNNEL );
    glPopMatrix();

    m_fpsc->countFramerate();
    m_rotate = m_fpsc->getFramerate();

    glFlush();
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key )
    {
        case 'x':
            viewer[0] -= 1.5;
            break;
        case 'X':
            viewer[0] += 1.5;
            break;
        case 'y':
            viewer[1] -= 1.5;
            break;
        case 'Y':
            viewer[1] += 1.5;
            break;
        case 'z':
            viewer[2] -= 1.5;
            break;
        case 'Z':
            viewer[2] += 1.5;
            break;
        case 27: //esc
            delete m_fpsc;
            exit(0);
        case 'j': //katse vasemmalle
            theta[ 1 ] -= 1.0;
            break;
        case 'k': //katse alas
            theta[ 0 ] += 1.0;
            break;
        case 'l': //katse oikealle
            theta[ 1 ] += 1.0;
            break;
        case 'i': //katse ylös
            theta[ 0 ] -= 1.0;
            break;
    }
    glutPostRedisplay();
}

void setup_GL( int w, int h )
{
    glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
    glutCreateWindow( "TunnelDemo" );
    glutFullScreen();

    //ladataan käytettävä tekstuuri
    char* filenames[] = { "ttimage.bmp" };
    for ( int i = 0; i < m_numTextures; i++ )
    {
        switch( loadOpenGL2DTextureBMP( filenames[i], &m_textureName[i] ) )
        {
            case LOAD_TEXTUREBMP_SUCCESS:
                break;
            case LOAD_TEXTUREBMP_COULD_NOT_FIND_OR_READ_FILE:
                exit(1);
            case LOAD_TEXTUREBMP_ILLEGAL_FILE_FORMAT:
                exit(1);
            case LOAD_TEXTUREBMP_OPENGL_ERROR:
                exit(1);
            case LOAD_TEXTUREBMP_OUT_OF_MEMORY:
                exit(1);
        }

        glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
        glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_NEAREST );
        glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
        glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
        glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
    }
}

```

```

glClearColor( 0.05, 0.05, 0.2, 0.0 );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_SMOOTH );

//Sumu
glFogi( GL_FOG_MODE, GL_LINEAR );
glFogfv( GL_FOG_COLOR, m_fog_color );
glFogf( GL_FOG_DENSITY, 0.5 );
glHint( GL_FOG_HINT, GL_DONT_CARE );
glFogf( GL_FOG_START, 0.0 );
glFogf( GL_FOG_END, 200.0 );
glEnable( GL_FOG );
}

void setup_projection()
{
    glMatrixMode( GL_PROJECTION );
    gluPerspective( 25.0, 1.44, 1.0, 1000.0 );

    glMatrixMode( GL_MODELVIEW );
    gluLookAt( 0.0, 0.0, 0.0,
              0.0, 0.0, -20.0,
              0.0, 1.0, 0.0);
}

void setup_light()
{
    GLfloat ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 15.0, 15.0, 15.0, 0.0 };
    GLfloat direction[] = { -1.0, -1.0, -1.0 };
    GLfloat lmodel_ambient[] = { 0.4, 0.4, 0.4, 1.0 };
    GLfloat local_view[] = { 0.0 };

    //Valot
    glLightfv( GL_LIGHT0, GL_AMBIENT, ambient );
    glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuse );
    glLightfv( GL_LIGHT0, GL_POSITION, position );
    glLightfv( GL_LIGHT0, GL_SPOT_DIRECTION, direction );
    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, lmodel_ambient );
    glLightModelfv( GL_LIGHT_MODEL_LOCAL_VIEWER, local_view );

    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
}

int main(int argc, char** argv)
{
    int width = GetSystemMetrics( SM_CXSCREEN );
    int height = GetSystemMetrics( SM_CYSCREEN );
    glutInit( &argc, argv );
    setup_GL( width, height );
    setup_light();
    setup_projection();
    make_tunnel();
    glutDisplayFunc( display );
    glutIdleFunc( display );
    glutKeyboardFunc( keyboard );
    m_fpsc = new FPSCounter();
    glutMainLoop();

    return 0;
}

```