Teemu Suopelto

TIETORAKENTEET JA ALGORITMIT NÄKYVYYSTARKASTELUSSA

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteet
Tietojenkäsittely
Syksy 2010

Kajaanin ammattikorkeakoulu

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteet	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Teemu Suopelto	
Työn nimi Tietorakenteet ja algoritmit näkyvyystarkastelussa	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Ohjaaja(t) Veli-Pekka Piirainen
	Toimeksiantaja Kajaanin Ammattikorkeakoulu / Pelilaboratorio
Aika Syksy 2010	Sivumäärä ja liitteet 41

Nykyaikaisten 3d-pelien grafiikka on äärimmäisen monimutkaista. Nopeasta kehityksestä huolimatta tietokonelaitteisto ei pysty vastaamaan kaikkein vaativimpien reaaliaikaisten sovellusten kuten pelien vaatimuksiin. Pelien 3d-grafiikka sisältää paljon pelaajalle näkymätöntä tietoa, jonka käsittely vie paljon aikaa. Tämän vuoksi pelisovelluksissa hyödynnetään näkyvyydentarkastelualgoritmeja, joiden tarkoituksena on piirtovaiheessa jättää käsittelemättä kuvaruudulla näkymätön data ja siten nopeuttaa kuvan tuottamista kuvaruudulle. Näkyvyystarkastelualgoritmit ovat olennainen osa jokaista nykyaikaista pelimoottoria.

Tämän opinnäytetyön tarkoitus on tutkia näkyvyystarkasteluissa käytettyjen tietorakenteiden ja algoritmien teoriaa. Lähteinä käytettiin akateemisia tutkimuksia ja grafiikkaohjelmointikirjoja. Aiheesta on olemassa erittäin paljon kirjoitettua tietoa, joten vain oleellisimpia asioita on käsitelty tässä opinnäytteessä. Teorian pääpaino on hierarkisissa tietorakenteissa, joiden lisäksi tutkittiin myös näkyvyyysoptimoinnin perusteita ja peittävyystarkastelua.

Taustateoria sovellettiin Kajak3D-pelimoottoriin, johon toteutettiin yleiskäyttöinen ja tehokas octree-lisäosa. Lisäosan vaatimusmäärittelyssä haluttiin tehostaa nimenomaan staattisten objektien piirtoa, ja sen tuli olla riittävän kevyt mobiiliympäristöön. Toteutuksessa haluttiin hyödyntää myös ennustettavuutta, jonka aikaisemmin tapahtuneiden näkyvyystarkasteluiden tutkiminen mahdollisti. Kajaanin amk:n pelilaboratorion henkilökunta osallistui vaatimusmäärittelyn tekemiseen ja auttoi Kajak3D:tä koskevissa teknisissä yksityiskohdissa.

Lisäosa testattiin Android-käyttöjärjestelmällä varustetulla älypuhelimella. Testi koostui Kajak3D:llä suoritettavasta 500-ruudun pituisesta kamera-animaatiosta kolmiuloitteisessa kaupunki-ympäristössä. Jokaisen ruudun prosessointiin ja piirtämiseen käytetty aika mitattiin. Tuloksia verrattiin piirtoon ilman octreeta. Piirto- ja prosessointiajan lisäksi mitattiin myös suoritettavien objektikohtaisten leikkaustestien määrä. Tulokset osoittivat, että octree-lisäosa paransi suorituskykyä huomattavasti. Myös leikkaustestien määrä väheni merkittävästi, koska octreen avulla pystyttiin hylkäämään isoja objektijoukkoja kerrallaan. Octree-lisäosa saavutti asetetut tavoitteet, ja se lisättiin osaksi Kajak3D-pelimoottoria.

Kieli	Suomi
Asiasanat	Näkyvyystarkastelu, peittävyystarkastelu, octree, tietorakenteet, algoritmit, pelit, kajak3d
Säilytyspaikka	☒ Verkkokirjasto Theseus ☒ Kajaanin ammattikorkeakoulun kirjasto

Kajaanin ammattikorkeakoulu
University of Applied Sciences

THESIS
ABSTRACT

School	Degree Programme
Natural Sciences	Information Technology
Author(s) Teemu Suopelto	
Title Data Structures and Algorithms in Visibility Determination	
Optional Professional Studies Game programming	Instructor(s) Veli-Pekka Piirainen
	Commissioned by Kajaani University of Applied Sciences
Date Fall 2010	Total Number of Pages and Appendices 41

Graphics in modern computer games are increasingly complex. Despite having very effective hardware, they simply cannot keep up with the demands of real-time applications such as games. At the same time 3d-graphics based games contain a lot of redundant information which takes a lot of time to process which is, however, never visible to the player. Some kind of visibility determination is often used to reduce the amount of processed graphical data and to speed up rendering. Visibility determination algorithms are an integral part of every modern game engine.

The purpose of this thesis is to study the theory behind basic data structures and techniques used in the visibility determination literature. The main sources of information are academic research papers and graphics programming books. A large body of research and data about visibility determination exists, thus only the most essential works have been included. The main part of the theory concerns hierarchical data structures, while the other parts consist of the basics of visibility optimization and occlusion culling.

The studied background theory was applied to a mobile game engine called Kajak3D and it was used to implement a practical, generic octree plug-in component including several enhancements. Requirements for the plug-in are specified to boost rendering of static objects in the mobile platforms. It was also required from the implementation that it utilizes temporal coherence data. The staff of the game development laboratory was involved in defining the requirements and they also provided technical support for Kajak3D.

Finally, the plug-in was tested on an Android-based mobile phone. The test scenario consists of a 500-frame long fly-through camera animation in a simplistic 3d-city environment which was rendered in real-time using Kajak3D. The time spent on processing and rendering each frame was saved and compared to the timing results without octree. Besides the processing and rendering time also a number of intersection tests were measured. The results of this work show that rendering with octree significantly increased performance. Also, the number of intersection tests was reduced significantly, because octree helped to reject groups of objects before rendering. The octree-plugin accomplished its objectives, and it was subsequently added to the Kajak3D component library.

Language of Thesis	Finnish
Keywords	Visibility determination, occlusion culling, octree, apmid, data structures, games, kajak3d
Deposited at	☒ Electronic library Theseus ☒ Library of Kajaani University of Applied Sciences

SISÄLLYS

SYMBOLILUETTELO

SYMBOLILUETTELO

ANDROID	Googlen kehittämä ja ylläpitämä käyttöjärjestelmä älypuhelimiin.
API	Application Programming Interface, ohjelmointirajapinta.
BADA	Samsungin kehittämä älypuhelin käyttöjärjestelmä.
BITANGENTTIVEKTORI	Pinnan normaalivektorin kanssa kohtisuorassa ja tekstuurikoordinaattien x-akselin suuntaan osoittava vektori.
DIRECTX	Microsoftin kehittämä grafiikkarajapinta ohjelmointia varten.
GPU	Graphics Processing Unit.
KONVEKSI	Kupera monikulmio, jonka kaikki sisäkulmat ovat korkeintaan 180 astetta, jolloin mikään kärkien yhdysjana ei käy kuvion ulkopuolella.
NORMAALIVEKTORI	Jonkin pinnan kanssa kohtisuorassa oleva vektori.
O-merkintä	Algoritmin suorituksen maksimiaikaa kuvaava merkintä.
OPENGL	Laitteistoriippumaton grafiikkarajapinta ohjelmointia varten.
ORIGO	Koordinaatiston alkupiste.
PIKSELI	Kaksiulotteinen kuvapiste.
POLYGONI	Umpinainen tasokuvio, joka koostuu janoista siten, että jokaisen janan päätepiste on jonkin toisen janan päätepiste. Janat eivät saa leikata tosiaan.
TANGENTTIVEKTORI	Pinnan normaalivektorin kanssa kohtisuorassa ja tekstuurikoordinaattien y-akselin suuntaan osoittava vektori.
VARJOSTIN	Näytönohjaimen prosessorilla suoritettava ohjelma.
VEKTORI	Suure, jolla on suunta ja suuruus.
VERTEKSI	Polygonin kulmapiste.

1 JOHDANTO

Varhaiset tietokonepelit pyrkivät esittämään kolmiulotteisia näkymiä vain muutamalla sadalla polygonilla. Aikoinaan nämä 3d-ympäristöt saattoivat tuntua hyvinkin mukaansa tempaavilta, mutta sillä viehätyksellä tuskin oli paljoakaan tekemistä grafiikan realismin kanssa. Polygonien lisääminen on tehnyt ympäristöistä vakuuttavampia, mutta samalla se on vaatinut laitteistolta parempaa suorituskykyä. Nykyaikaiset näytönohjaimet pystyvät piirtämään jo useita miljoonia kolmioita reaaliaikaisten pelien vaatimalla 60 ruudun sekuntivauhdilla, mutta edelleen suunnittelijat joutuvat tasapainoilemaan realismin ja nopeuden välillä. Pelien fotorealismiin pyrkivä ympäristöjen monimutkaisuus vaatii laitteistolta paljon enemmän laskentatehoa, kuin se vielä pitkään aikaan pystyy tarjoamaan.

Tehokkuutta voidaan kuitenkin lisätä huomattavasti erilaisten optimointien avulla, joista keskeisin koskee kappaleiden näkyvyyttä. On tavallista, että monet esineet ovat kokonaan tai osittain toisten esineiden takana piilossa, joten niiden käsittelyyn ja piirtämiseen käytetty aika on hukkaan heitettyä, koska tälläiset osat eivät kuitenkaan näy katsojalle. Piilossa olevien kappaleiden etsimistä varten on vuosien saatossa pyritty kehittämään useita erilaisia näkyvyystarkastelutekniikoita ja -algoritmeja. Mitään täysin yleistä ja universaalia tapaa siihen ei vieläkään ole olemassa, mutta olemassa olevia tekniikoita oikein soveltamalla voidaan saada hyviä tuloksia.

Näkyvyystarkastelusta on tullut olennainen osa jokaista nykyaikaista pelimoottoria ja se on tärkein yksittäinen tapa reaaliaikaisen piirron nopeuttamiseksi. Luonnollisesti myös Kajaanin Ammattikorkeakoulun Pelilaboratorion kehittämä Kajak3D-pelimoottori tarvitsee tehokkaan näkyvyysalgoritmitoteutuksen. Kajak3D-pelimoottori on suunniteltu mobiilialustoja silmällä pitäen, joissa laskennalliset resurssit ovat hyvinkin vaatimattomat verrattuna PC-tietokoneisiin.

Tässä opinnäytetyössä perehdytään reaaliaikaisen 3d-grafiikan perusteisiin sekä keskeisiin näkyvyystarkastelun algoritmeihin ja tietorakenteisiin, joiden pohjalta on toteutettu Kajak3D:hen staattisten objektien octree-lisäosa.

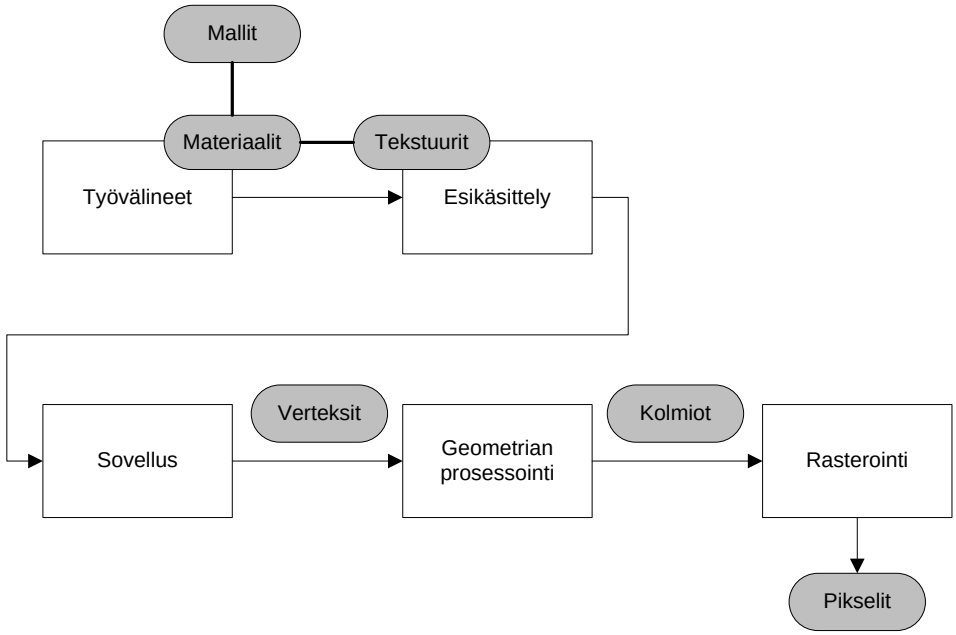
2 3D-PIIRRON JA NÄKYVYYSTARKASTELUN PERUSTEET

Reaaliaikainen 3d-piirto voidaan karkeasti jakaa seuraaviin perusosiin: Virtuaalinen ympäristö, joka koostuu jossakin matemaattisessa muodossa kuvatuista kolmiulotteisista pinnoista. Ympäristöön on määritelty muutamia valonlähteistä, jotka valaisevat pintoja. Pinnoilla on tiettyjä visuaalisia ominaisuuksia eli materiaaleja, jotka määrittelevät miten valo vuorovaikuttaa kunkin pinnan kanssa. Ympäristöä katsotaan tai kuvataan virtuaalisella kameralla, joka on asetettu ympäristön johonkin kohtaan osoittamaan tiettyyn suuntaan.

Käytännössä lähes aina reaaliaikaisessa 3d-grafiikassa käsiteltävät pinnat ovat kolmisivuisia polygoneja (eli kolmioita), jotka muodostavat virtuaaliympäristön objektit. Objektien ja valojen ominaisuudet ovat määritelty ohjelmoitavien varjostimien avulla. (Gregory 2009, 400.) Valonlähteiden ja pintojen tuottamaa kuvaa, joka piirretään kameran näkökentän kautta, kutsutaan piirtoyhtälön tai valaistusyhtälön ratkaisemiseksi. Tätä yhtälöä toistetaan 30 tai 60 kuvan sekuntivahdilla uskottavan liikeilluusion aikaansaamiseksi. Tämä tarkoittaa, että renderöijällä on vain 33.3 tai 16.6 millisekuntia aikaa tuottaa lopullinen kuva. Käytännössä aikaa on vielä vähemmän, sillä animaatio, tekoäly ja muut osa-alueet kuluttavat siitä myös osan. (Gregory 2009, 401.)

2.1 3D-piirron liukuhihna

Ohjelmisto- ja laitteistoarkkitehtuuria, jolla reaaliaikainen 3d-grafiikan piirto on toteutettu, kutsutaan piirron liukuhihnaksi (engl. rendering pipeline). Nimitys johtuu siitä, että piirrettävänä olevan objektin geometria- ja materiaalitieto kulkee useiden toisistaan riippumattomien vaiheiden kautta. Jokaisella vaiheella on oma määritelty tarkoituksensa ja vaihe jalostaa (eli transformoi) tietoa ja välittää sen eteenpäin seuraavalle vaiheelle, kunnes se lopulta piirretään pikseleinä kaksiulotteiselle kuvaruudulle. (Puhakka 2008, 163.) Kuviossa 1. havainnollistetut 3d-piirron liukuhihnan päävaiheet ovat seuraavat:



Kuvio 1. Piirron liukuhihna ja miten tieto muuttuu eri vaiheiden välillä (Mukaillen Gregory 2009, 446).

Työvälinevaihe (engl. tools stage), jossa taiteilijat luovat mallit ja tekstuurit digitaalisilla sisällöntuotanto ohjelmilla kuten esimerkiksi Autodesk Maya, Autodesk 3ds Max, Newtek Lightwave tai Adoben Photoshop. Artistit määrittelevät myös materiaalit ja parametrit materiaalivarjostimia varten materiaalieditorilla. Materiaalieditori voi olla lisäosa 3d-mallinnusohjelmaan tai erillinen sovellus kuten Nvidian Fx Composer. (Gregory 2009, 447.)

Resurssien esikäsittelyvaihe (engl. asset conditioning stage), jossa geometria- ja materiaalitieto puretaan sisällöntuotantosovellukselta ja tallennetaan yleensä alustariippumattomaan välimuotoon. Tämän jälkeen data prosessoidaan yhteen tai useampaan alustakohtaiseen muotoon, riipuen siitä kuinka montaa eri alustaa ja laitteistoa pelimoottori tukee. Esimerkiksi 3d-mallidata saatetaan prosessoida muotoon, josta se voidaan lukea Xbox 360 ja Playstation 3 –konsolien videomuistiin mahdollisimman nopeasti. Resurssien esikäsittelyvaihe joutuu myös yleensä ottamaan huomioon materiaalien ja varjostimien vaatimukset. Esimerkiksi jos jokin varjostin tarvitsee tangenttivektori- ja bitangenttivektoritiedot, niin ne voidaan generoida tässä vaiheessa. Myös kaikenlainen muu esikäsittely tehdään tässä vaiheessa, kuten valaistuskartan (engl. light map) laskeminen tai BSP-puun (ks. kappale 4.9) rakentaminen kenttägeometriasta. (Gregory 2009, 447.)

Sovellusvaiheen (engl. application stage) tehtävä on valmistella piirrettävänä oleva malli näytönohjaimelle lähettämistä varten. Sovellusvaiheen tarkka sisältö riippuu aina piirron käyttötarkoituksesta, mutta yleensä sovellusvaihe on vastuussa ainakin näkyvyystarkistuksista, geometrian lähettämisestä näytönohjaimelle ja varjostinparametrien kontrolloimisesta. Sovellusvaiheen tehtävät hoidetaan prosessorilla.

Geometriavaiheessa (engl. geometry processing stage) piirrettäville kolmioille ja kolmion kulmapisteille, eli vertekseille, tehdään geometrisia muunnoksia, jonka lopputuloksena malli voidaan lähettää piirrettäväksi rasterointivaiheelle. Muunnokset maailman- ja kamerankoordinaatistoon tehdään tässä vaiheessa, kuten myös valaistuksenlaskenta. Nykyaikaisella laitteistolla nämä muunnokset lasketaan ohjelmoitavilla verteksivarjostimilla (engl. vertex shader). Verteksivarjostimet suoritetaan näytönohjaimen prosessorilla, eli GPU :lla. (Puhakka 2008, 168.)

Rasterointivaihe (engl. rasterization stage) on piirron viimeinen vaihe. Geometriavaiheen jälkeen piirretävä tieto on muunnettu ruudun koordinaatistoon, ja tässä vaiheessa siitä tehdään lopullinen kaksiulotteisista pikseleistä koostuva rasterikuva. Yleensä geometria tekstuuroidaan, eli kolmioiden pinnalle piirretään jokin kaksiulotteinen kuva ja aivan rasteroinnin loppuvaiheessa pikseleille tehdään muutamia testejä (kuten Z-testi jossa tutkitaan jääkö pikseli jonkin toisen kappaleen taakse piiloon), jotka määräävät piirretäänkö pikseli lopulliseen kuvaan. Myös rasterointivaihe hoidetaan nykyään ohjelmoitavien pikselivarjostimien (engl. pixel shader, fragment shader) avulla, ja vaihe suoritetaan näytönohjaimen prosessorilla. (Puhakka 2008, 169.)

Tämän opinnäytteen kannalta kiinnostavin ja tärkein vaihe on tietenkin sovellusvaihe, johon näkyvyystarkistukset sijoittuvat. Monet algoritmit, kuten esikäsittelyyn perustuvat PVS- tai syvyyspuskuriatietoja käyttävät peittävyystarkistelut, hyödyntävät myös muita vaiheita.

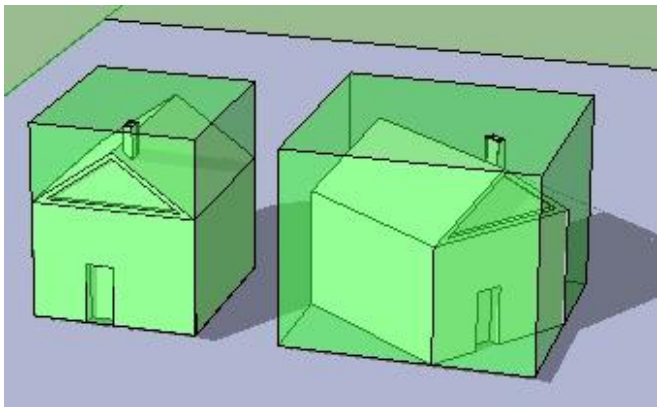
2.2 Rajauslaatikot

Näkyvyystarkasteluissa joudutaan usein tekemään erilaisia tarkistuksia ja geometrisia testejä, kuten onko jokin objekti toisen takana, tai leikkaako objekti kameran näkökenttää. Kaikki laskelmat voitaisiin tehdä käyttämällä kappaleen varsinaista polygoneista koostuvaa geomet-

riaa, jolloin näkyvyystarkastelu olisi äärimmäisen tarkka. Näin ei käytännössä kuitenkaan yleensä tehdä, koska kolmioista koostuvan kappaleiden muoto on yleensä hyvin monimutkainen ja tarkistusten laskeminen olisi erittäin hidasta. Tämän vuoksi kappale ympäröidään yksinkertaisemmalla geometrisella muodolla (Kuvio 2.), jota kutsutaan rajauslaatikoksi (engl. bounding volume). Laskelmat tehdään käyttämällä varsinaisen geometrian sijaan rajauslaatikkoa.

Usein käytettyjä rajauslaatikoita ovat esimerkiksi koordinaattiakselien suuntainen suorakulmio (engl. axis-aligned bounding box) ja pallo. Hyviä rajauslaatikon ominaisuuksia ovat:

- edulliset leikkaustestit
- ympäröidä alkuperäinen malli mahdollisimman tiiviisti
- nopea laskea
- tarvitsee vähän muistia



Kuvio 2. Rajauslaatikot (Slava).

Rajauslaatikoita ei yleensä lasketa ajon aikana, vaan esikäsittelyvaiheessa. Joitakin rajauslaatikoita täytyy muuttaa tai tasata myös ajon aikana, mikäli niiden ympäröimä objekti on liikkunut. Koska rajauslaatikot tallennetaan varsinaisen geometrian rinnalle, on suotuisaa että rajauslaatikko muoto kuluttaa mahdollisimman vähän muistia. (Ericson 2005, 76.)

2.2.1 Koordinaattiakselien suuntainen suorakulmainen särmiö

Koordinaattiakselien suuntainen suorakulmainen särmiö (Kuvio 2.), eli AABB (axis-aligned bounding box) on yksinkertainen kuusisivuinen laatikko, jonka sivut ovat pääakselien suuntaiset. Laatikon ei tarvitse olla kuutio, vaan sen sivujen pituus, korkeus ja leveys voivat vaihdella. Kaikki laatikon sisäpuolelle jäävät pisteet toteuttavat seuraavat epäyhtälöt:

$$x_{min} \leq x \leq x_{max}$$

$$y_{min} \leq y \leq y_{max}$$

$$z_{min} \leq z \leq z_{max}$$

Suorakulmio esitetän yleensä kahden kulmapisteen avulla. Kulmapisteet P_{min} ja P_{max} muodostuvat koordinaattien minimi- ja maksimiarvoista:

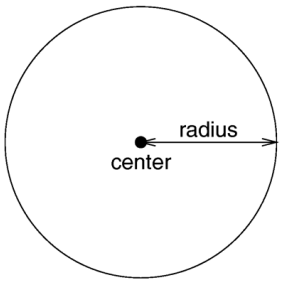
$$P_{min} = [X_{min}\ Y_{min}\ Z_{min}]$$

$$P_{max} = [X_{max}\ Y_{max}\ Z_{max}]$$

Minimi- ja maksimimikulmapisteiden lisäksi muita suosittuja AABB esitystapoja ovat muun muassa esitys keskipisteen ja sädevektorin avulla, sekä esitys kulmapisteen ja laatikon läpimitan ilmaisevan vektorin avulla. (Dunn & Parberry 2002, 248.)

2.2.2 Pallo

Pallo on kolmiulotteinen objekti joka määritellään joukkona pisteitä, jotka ovat tietyllä etäisyydellä tietystä pisteestä. Etäisyyttä pallon keskipisteestä pisteeseen kutsutaan pallon säteeksi. Pallo esitetään koordinaattipisteenä **c**, joka merkitsee pallon keskipistettä ja skalaariarvolla *r*, joka on pallon säde (Kuvio 3.). (Dunn & Parberry 2002, 246.)



Kuvio 3. Pallo (havainnollistettu ympyrällä) määriteltynä pisteen ja säteen avulla (Povray).

Pallo on AABB:n lisäksi hyvin yleinen rajausmuoto, jonka leikkaustestit ovat hyvin nopeita. Pallo ei myöskään muuta muotoaan vaikka sitä pyöritettäisiin, joten samaa palloa voidaan käyttää rajausmuotona vaikka rajatun objektin kierto muuttuisikin. Optimaalisen rajauspallon laskeminen ei ole aivan niin helppoa kuin laatikon laskeminen, mutta riittävän hyvän likiarvon laskemiseen on kehitetty useita tapoja.

2.2.3 Taso

Taso on täydellisen litteä pinta 3d-avaruudessa, sillä ei ole paksuutta ja se on ääretön. Tasoa itsessään ei voida käyttää rajausmuotona, mutta sitä tarvitaan monissa leikkaustarkituksissa ja testeissä. Eräs suosittu tapa esittää tason yhtälö on karteesinen muoto, joka koostuu normaalivektorista **n** = [A, B, C], sekä pisteestä **p** = [x, y, z] tasolla (Dunn & Parberry 2002, 252). Tällöin tason yhtälöksi saadaan:

$$Ax + By + Cx = d \qquad (1)$$

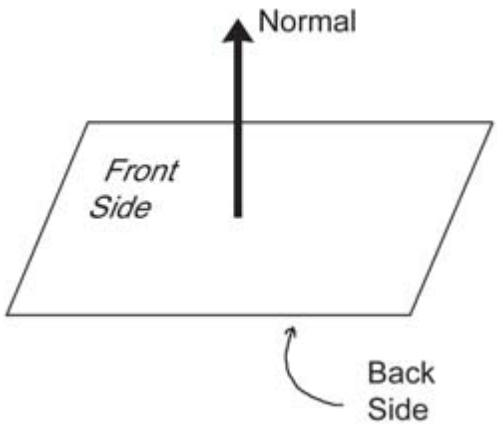
$$\mathbf{p} \cdot \mathbf{n} = d$$

Vektoria **n** kutsutaan tason normaalisiksi, koska se on kohtisuora tason pintaan nähden. Voidaan sanoa että tasolla kaksi on puolta, etupuoli ja takapuoli (Kuvio 4.). Yleensä etupuoleksi kutsutaan suuntaa johon vektori **n** osoittaa (Dunn & Parberry 2002, 253). Tason yhtälö saatetaan esittää joissain lähteissä muodossa:

$$Ax + By + Cz + D = 0 \qquad (2)$$

Jossa D saadaan kaavalla:

$$D = -\mathbf{p} \cdot \mathbf{n} \qquad (3)$$



Kuvio 4. Tason etu- ja takapuoli, sekä tason normaali (Dunn & Parberry 2002, 253).

Usein halutaan tutkia kuinka kaukana ja kummalla puolella tasoa jokin piste sijaitsee. Pisteen Q = [X, Y, Z] etäisyys tasosta saadaan kaavalla 4.

$$a = \frac{AX+BY+CZ+D}{\sqrt{A^2+B^2+C^2}} \qquad (4)$$

Tästä saatu etäisyys *a* on etumerkillinen, eli jos etäisyys on positiivinen, sijaitsee piste tason normaalin osoittamalla puolella, jos taas negatiivinen niin piste on tason takapuolella. Mikäli etäisyys on 0, niin piste sijaitsee tasolla. (Puhakka 2008, 44.)

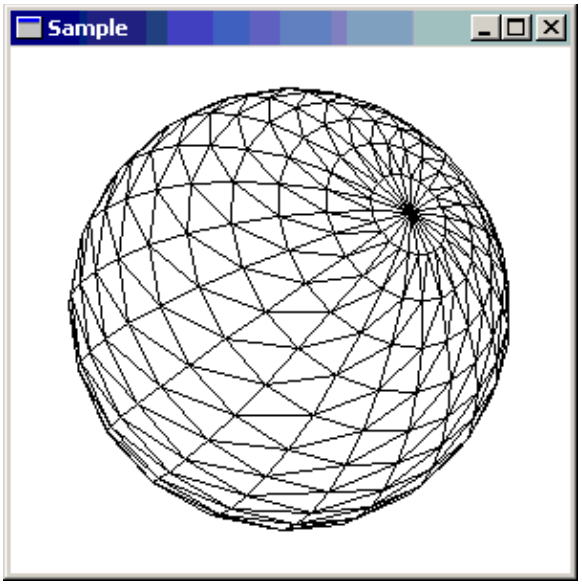
2.3 Z-puskuri ja Z-testi

Syvyyspuskuri (engl. Z-buffer) on koko ruudun kokoinen puskuri, joka tyypillisesti sisältää 16- tai 24-bittisen syvyystiedon jokaiselle pikselille. Ennen kuin pikseliä ollaan piirtämässä ruudulle, tallennetaan sen syvyysarvo syvyyspuskuriin. Tätä tietoa tarvitaan päättelemään mi-

kä objekti on minkäkin objektin edessä. Kun toista pikseliä jostain toisesta mallista ollaan piirtämässä ruudulle samaan kohtaan, verrataan uuden pikselin syvyysarvoa syvyyspuskurissa jo olevaan arvoon. Jos uusi pikseli on lähempänä kameraa (eli sen syvyysarvo on pienempi), ylikirjoittaa se vanhan arvon syvyyspuskurissa. Tätä kutsutaan Z-testiksi (engl. Z-test). Z-testi tehdään rasteroinnin loppuvaiheessa, mutta jotkin uudet näytönohjaimet suorittavat Z-testin varhaisemmassa vaiheessa (engl. early z-testing) ennen kuin pikselit lähetään pikselivarjostimille. (Gregory 2009, 443.)

2.4 Takapintojen poisto

Takapintojen poistossa (engl. backface culling) ideana on välttää piirtämästä sellaista geometriaa, joka ei ole suunnattu kameraa kohti. Esimerkiksi kamerassa olevassa 3d-hahmon polygoneista noin puolet osoittavat kohti kameraa, ja puolet polygoneista pois päin kamerasta. Esimerkiksi, jos hahmo katsoo kameraan päin, niin hahmon selkäpuoli ei näy kameraan. Takapintojen poistossa tälläiset kamerasta pois päin suunnatut polygonit jätetään piirtämättä (Kuvio 5.).



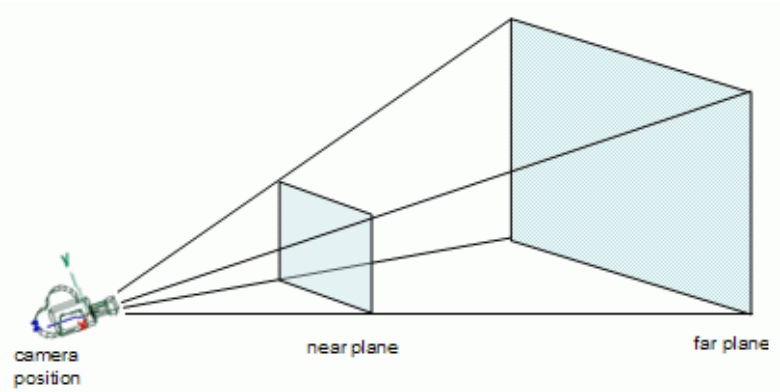
Kuvio 5. Toisella puolella olevia kolmioita ei ole piirretty (GPWiki).

Suunta johon polygoni osoittaa, lasketaan kameran suunnasta ja polygonin normaalista. Takapintojen poisto toteutetaan OpenGL ja DirectX -grafiikkarajapintojen toimesta automaat-

tisesti, ja lisäksi käyttäjä voi määritellä haluaako jättää piirtämättä polygonien etu- vai takapinnat, vai molemmat. (Sherrod 2008, 529.)

2.5 Näkökartion ulkopuolelle jäävien objektien poisto

Näkökartio (engl. view frustum) on alue, joka muodostaa kameran näkökentän. Kaikki näkökartion sisäpuolelle jäävät objektit ovat näkyvissä ruudulla (pois lukien mahdolliset toisensa peittävät objektit). Näkökartion tarkka muoto on riippuvainen kamera projektion asetuksista, mutta perspektiivikamerassa kartio on kutakuinkin katkaistun pyramidin muotoinen (Kuvio 6.). (Fernandes, A. R.)



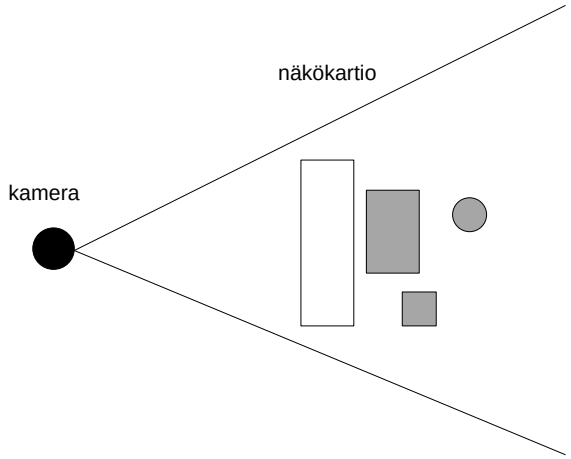
Kuvio 6. Näkökartio (Fernandes, A. R).

Pyramidi muodostuu kuudesta leikkaustasosta. Pyramidin huippupiste on kameran paikka ja pyramidin pohjaa kutsutaan takaleikkaustasoksi (engl. far plane). Pyramidi on katkaistu etuleikkaustason (engl. near plane) kohdalta, jolloin se muistuttaa kartiota (Fernandes, A. R). Tarkistus, jolla tutkitaan leikkaako objektin rajauslaatikko näkökartiota, on melko yksinkertainen. Perusidea on tarkistaa sijoittuvatko laatikon kaikki kulmapisteet jonkin leikkaustason takapuolelle. Tarkistukset tehdään yksitellen jokaiselle näkökartion kuudelle leikkaustasolle. Jos kaikki kulmapisteet ovat yhdenkään leikkaustason takapuolella, laatikko on kartion ulkopuolella eikä sitä tarvitse piirtää. On huomattava, että tämä leikkaustesti ei ole täysin tarkka, vaan antaa väärän tuloksen joillekin taka- ja sivuleikkaustasoa leikkaaville objekteille, vaikka objekti ei olisikaan näkyvissä. Tämä saattaa joskus johtaa piilossa olevien objektien turhaan piirtämiseen ja on tapauskohtaisesti arvioitava onko tarkemmasta tarkastelusta hyötyä. Tar-

kistuksen tarkoitus on vähentää näytönohjaimelle lähetettävien objektien määrää. Tämä on perusnäkyvyysoptimointi, jota pyritään nopeuttamaan hierarkisilla tietorakenteilla, kuten oct-reella. (Dunn & Parberry 2002, 387.)

3 PEITTÄVYYSTARKASTELU

Vaikka objekti olisikin täysin kameran näkökartion sisäpuolella, jokin toinen objekti saattaa peittää sen niin, ettei sitä käytännössä näy lopullisessa kaksiulotteisessa kuvassa ollenkaan (Kuvio 7.). Tälläisten piilossa olevien objektien havaitsemista ja poistamista renderöintilistasta kutsutaan peittävyystarkasteluksi (engl. occlusion culling). Ruuhkaisissa ja tiheissä ympäristöissä saattaa peittävyyttä olla hyvinkin paljon, jolloin peittävyystarkastelusta on huomattavaa hyötyä. Vastaavasti taas tilavissa ja avarissa kentissä peittävyystarkistukset saattavat olla turhia ja vaikuttaa jopa negatiivisesti suorituskykyyn. (Gregory 2009, 461.)



Kuvio 7. Kaikki objektit ovat näkökartion sisäpuolella, mutta harmaat objektit ovat peitettyinä valkoisen objektin takana.

Peittävyystarkastelu algoritmit voidaan jakaa ruudunkoordinaatisto (image space), objektikoordinaatisto (object space) ja sädekoordinaatisto (ray space) algoritmeihin. Ruudunkoordinaatisto algoritmit tekevät näkyvyystestejä kaksiulotteisen kameraprojektion jälkeen, kun taas objektikoordinaatistossa toimivat algoritmit käyttävät tarkasteluihin alkuperäisiä 3d-objekteja. Sädekoordinaatisto algoritmit taas hyödyntävät molempia avaruuksia, usein niin että kaksiulotteinen kuvapiste muunnetaan säteeksi, kuten raycastingissa. (Akenine-Moller, Haines & Hoffman 2008, 672.)

Tässä luvussa esittelemme kaksi peittävyystarkastelutekniikkaa. Molemmat sopivat parhaiten sisätiloille, joissa seinät, katto ja lattiat peittävät isoja alueita. Molempia yleensä käytetään yh-

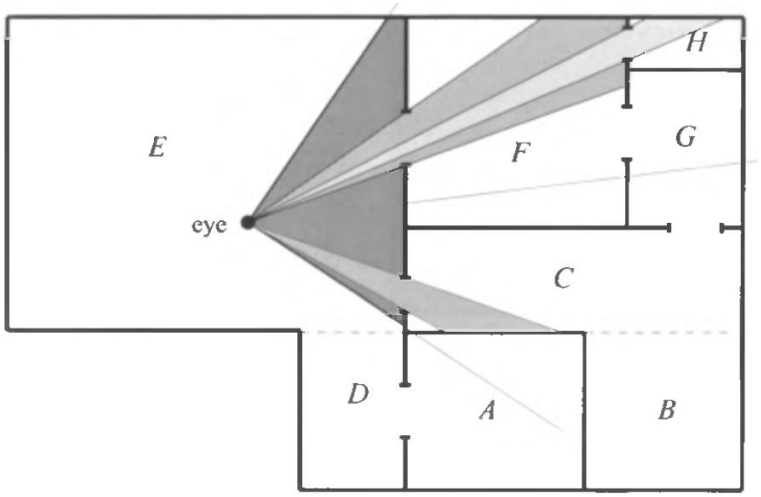
dessä jonkin spatiaalisen tietorakenteen kanssa ja molempia tekniikoita voidaan käyttää yhdessä.

3.1 Portaalit

Portaalikarsinta (engl. portal culling, portal rendering) on sisätiloille tarkoitettu peittävyystarkastelutekniikka. Perustana on ympäristön jako konveksin muotoisiin soluihin, jotka ovat yhdistetty toisiinsa portaaleilla. Solut vastaavat ympäristön huoneita ja käytäviä. Portaalit ovat huoneita yhdistäviä ovia ja ikkunoita. (Akenine-Moller, Haines & Hoffman 2008, 667.)

Konveksi-muoto on tärkeä, koska koko solu on aina näkyvissä jokaisesta solun sisällä olevasta pisteestä. Tämä helpottaa näkyvyystarkastelua, koska tällöin näkyvyyttä ei tarvitse enää tutkia solun sisäpuolella. (Puhakka 2008, 272.)

Jokainen solu sisältää listan solun sisäpuolella olevista objekteista sekä solun rajaavista seinämistä. Lista soluista ja niitä yhdistävistä portaaleista tallennetaan naapuruusgraafiin (engl. adjacency graph). Graafi luodaan esikäsittelyvaiheessa, vaikka se periaatteessa on mahdollista generoida automaattisesti, modernien 3d-ympäristöjen automaattinen prosessointi on erittäin vaikeaa ja siksi graafi kannattaa tehdä käsin. (Akenine-Moller, Haines & Hoffman 2008, 667.)



Kuvio 8. Kameran näkökartio ja solut (Akenine-Moller, Haines & Hoffman 2008, 669).

Kun tiedossa on konveksisolut, solut toisiinsa liittävä naapuruusgraafi, katsojan sijainti sekä katsojan näkökartio, voidaan selvittää mitkä solut ovat näkyvissä. Sisätiloissa seinät, lattia ja katto estävät näkymisen suurimpaan osaan näkökartion sisäpuolelle jäävistä objekteista, joten muihin soluihin näkee vain portaalien kautta. (Puhakka 2008, 277.)

Ensiksi on etsittävä katsojan sijainnin ja naapuruusgraafin avulla kaikki portaalit, jotka leikkaavat näkökartiota. Jokaista kartiota leikkaavaa portaalia varten lasketaan uusi näkökartio, ja piirretään se solu johon portaali johtaa (joka selviää naapuruusgraafista), toistaen prosessi rekursiivisesti. Usein näkökartio kapenee levitessään uuteen soluun. Tämä on havainnoillistettu kuviossa 8., jossa toisiin soluihin leviävän näkökartion osat on merkitty vaalean harmaalla. Kamera sijaitsee solussa E. Naapurisoluja ovat solut C, D ja F. Alkuperäinen näkökartio ei kuitenkaan näe soluun D, joten sitä ei tarvitse tutkia pidemmälle. Jokaisesta solusta ei tietenkään tarvitse piirtää kuin näkökartiota leikkaavat seinät ja objektit. Portaalien leikkaustesteissä voidaan käyttää samaa tarkistusta kuin normaaleille objekteillekin. (3D Kingdoms 2006.)

3.2 Potentially Visible Set

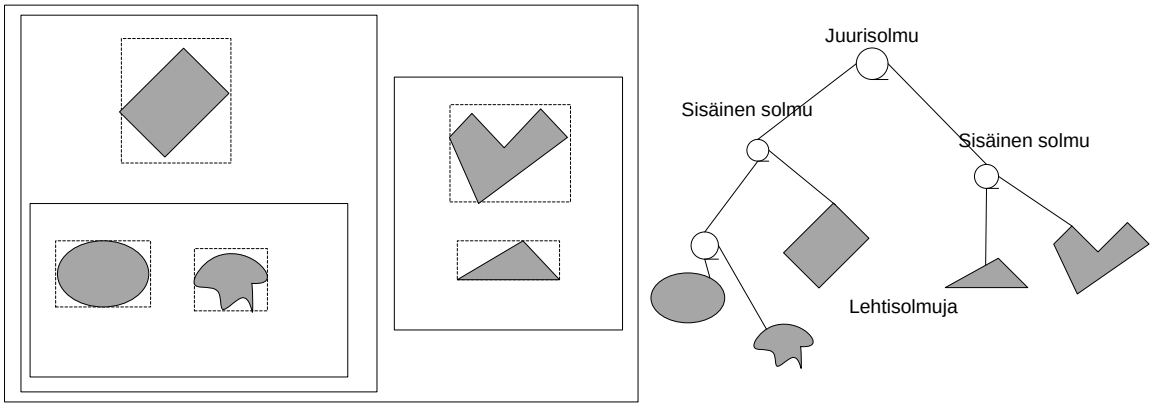
PVS, eli Potentially Visible Set –pohjaiset tekniikat yrittävät ratkaista peittävyyden ongelman esikäsittelynä. Perusidea on, että kun ympäristö on jaettu soluihin jollain muulla tekniikalla (kuten octreella tai BSP-puulla), voidaan jokaisesta solusta tarkastella mitkä muut solut, objektit tai pinnat, ovat näkyvissä solun jokaisesta pisteestä, ja tehdä niistä lista. Tämä lista, eli PVS, tallennetaan soluun. Ympäristöä piirrettäessä tarvitsee vain selvittää missä solussa kamera sillä hetkellä sijaitsee, ja lukea lista näkyvistä kohteista. Listaan voidaan lisäksi soveltaa perinteistä näkökartion leikkaustarkistusta, ja piirtää ainoastaan objektit ja alueet jotka ovat kameran näkökentässä. (Carter 2004, 233.)

Potentiaalisesti näkyvien joukkojen rakentamiseen on useita tekniikoita, ja ne ovat usein hyvin aikaa vieviä. Prosessin nopeuttamiseksi ei useinkaan lasketa täydellistä PVS:sää, vaan jonkinlainen hyvä arvio usein riittää. Seth Teller on väitöskirjassaan (Teller 1992) esitellyt hyvin kattavasti eri tekniikoita Potentially Visible Set –laskentaan.

4 RAJAUSLAATIKKOHIERARKIAT

Rajauslaatikoiden käyttäminen objektien varsinaisen geometrian sijaan nopeuttaa törmäystarkistuksia huomattavasti, mutta testien lukumäärä pysyy silti samana. Järjestämällä rajauslaatikot puu-hierarkiaan, saadaan testien lukumäärä vähennettyä huomattavasti. Tälläistä tietorakennetta kutsutaan rajauslaatikkohierarkiaksi (engl. bounding volume hierarchy). (Ericson 2005, 235.)

Puun rakenne koostuu juurisolmusta (engl. root node), sisäisistä solmuista (engl. internal nodes) sekä lehtisolmuista (engl. leaf nodes). Juurisolmu on hierarkiassa ylimmäisenä, eikä sillä ole vanhempia (engl. parent node). Sisäisillä solmuilla on osoittimet lapsisolmuihin, jotka voivat olla joko sisäisiä solmuja tai lehtisolmuja. Lehtisolmut sisältävät ympäristön geometrian, eikä niillä ole lapsisolmuja. Jokaisella puun solmulla (mukaan lukien lehtisolmut) on rajauslaatikko, joka rajaa itsensä ja solmun lapsien rajauslaatikot. Tämä tarkoittaa, että juurisolmulla on rajauslaatikko, joka pitää sisällään koko ympäristön. Kuviossa 9. on esitelty yksinkertainen kuuden objektin rajauslaatikkohierarkia. (Akenine-Moller, Haines & Hoffman 2008, 648.)



Kuvio 9. Kuuden objektin rajauslaatikkohierarkia (Mukaillen Ericson 2005, 236).

Rajauslaatikoiden organisoiminen puurakenteeseen on hyödyllistä, koska jos puun jonkin solmun rajauslaatikko ei ole lainkaan näkyvissä, niin silloin mikään sen lapsisolmun rajauslaatikoista ei myöskään ole näkyvissä. Silloin tarkistukset lapsisolmujen kanssa voidaan jättää tekemättä.

4.1 Hyvän rajauslaatikkohierarkian ominaisuuksia

Aivan kuten rajauslaatikollekin, myös niistä koostuville hierarkioille on määritelty muutamia suotuisia ominaisuuksia (Ericson 2005, 236-237):

- Mitä alempana solmu on puussa, sitä lähempänä sen tulisi sijaita sen sisarsolmuja.
- Jokaisen solmun rajauslaatikko pitäisi olla mahdollisimman tiivis. Tiiviimmät rajauslaatikot mahdollistavat tarkemmat näkyvyystarkistukset.
- Puun rajauslaatikoiden summa tulisi olla mahdollisimman pieni.
- Sisarsolujen päällekkäisyyksiä tulisi olla mahdollisimman vähän.
- Hierarkian tulisi olla tasapainoinen. Tasapainoinen puu mahdollistaa turhien haarojen hylkäämisen aikaisin, jolloin vältytään turhilta testeiltä.

Stefan Gottschalk muotoili artikkelissaan (Gottschalk, 1996) seuraavan funktion arvioidakseen rajauslaatikkohierarkiaan tehtävien hakujen kustannuksia:

$$T = N_v * C_v + N_p * C_p$$

Em. funktiossa T on haun kustannus, N_v on päällekkäisten rajauslaatikoiden lukumäärä, C_v on rajauslaatikkoparin päällekkäisyyden testauskustannus, N_p on objektiparien leikkaustestien lukumäärä ja C_p on objektiparin leikkaustestin kustannus. Näistä muuttujista N_v ja N_p saadaan pienennettyä pitämällä rajauslaatikot mahdollisimman tiiviinä ja C_v ja C_p saadaan minimoitua tekemällä leikkaustestit mahdollisimman nopeiksi. Valitettavasti nämä ovat usein ristiriidassa, sillä usein tiivis rajauslaatikkomuoto tekee leikkaustesteistä hitaampia. Esimerkiksi pallon leikkaustesti on jonkin verran laatikon leikkaustestiä nopeampi, mutta usein pallo vastaa hyvin huonosti objektin alkuperäistä muotoa (Gottschalk, 1996). Gottschalkin kustannusfunktio on alunperin tarkoitettu törmäystarkistuksia varten, mutta sitä voidaan soveltaa myös näkyvyystarkasteluun. (Ericson 2005, 237-238.)

4.2 Hierarkian rakentaminen ylhäältä-alas metodilla

Ylhäältä-alas (engl. top-down) metodi on rekursiivinen. Se aloitetaan laskemalla joukolle objekteja yhteinen rajauslaatikko. Tämä objektien joukko jaetaan kahteen alijoukkoon (tai k-määrään alijoukkoja), joille kummallekin lasketaan omat rajauslaatikot. Tätä toistetaan rekursiivisesti, kunnes joukkoa ei voida enää jakaa pienempiin osiin tai jokin muu ennalta määrätty ehto, kuten joukon objektien minimilukumäärä tai hierarkian maksimisyvyys, täyttyy. (Ericson 2005, 240.)

Joukon jakamista varten valitaan sopiva akseli minkä mukaan joukko halkaistaan, ja josta etsitään hyvä halkaisupiste. Yksinkertainen ja hyvän tuloksen tuottava tapa on valita akseli, joka kulkee joukon rajauslaatikon kauimpana toisistaan olevien pisteiden läpi. Muita yleisiä tapoja on esimerkiksi valita pisin lokaali x, y tai z –akseli. (Ericson 2005, 241-242.)

Halkaisupisteeksi valitaan yleensä joko keskimmäisen objektin keskikohta (object median) jolloin objektit jakaantuvat tasaisesti alijoukkojen kesken, tai koordinaatiston keskikohta (object mean) tai rajauslaatikon keskikohta (spatial median). Rajauslaatikon keskikohta on houkutteleva vaihtoehto, koska se voidaan laskea vakioajassa O(1) suoraan rajauslaatikosta, ilman että sen sisältöä tarvitsee tarkastella. (Ericson 2005, 244.)

Ylhäältä-alas metodi on varmasti yleisin tapa rajauslaatikkohierarkioiden rakentamiseen. Sen vahvuuksia ovat helppo toteutettavuus ja nopea puun rakennusaika. Ylhäältä-alas rakennetut puut eivät kuitenkaan ole optimaalisia. (Ericson 2005, 245.)

4.3 Hierarkian rakentaminen alhaalta-ylös

Alhaalta-ylös puun rakentaminen aloitetaan rajaamalla ympäristön jokainen objekti rajauslaatikolla. Nämä rajauslaatikot ovat puun lehtisolmuja. Lehtisolmuista valitaan kaksi jonkin tietyn valintakriteerin mukaan, jotka sitten ympäröidään isommalla rajauslaatikolla. Näin jatketaan kunnes tuloksena on yksi iso rajauslaatikko, joka on myös hierarkiapuun juurisolmu. (Ericson 2005, 245.)

Pareiksi valitaan yleensä sellaiset, jotka ovat lähimpänä toisiaan, jolloin ympäröivä rajauslaatikko saadaan mahdollisimman pieneksi. Alhaalta-ylös rakentaminen on hieman hitaampaa ja monimutkaisempaa kuin ylhäältä alas rakentaminen, mutta hierarkiat ovat yleensä optimaalisempia. (Ericson 2005, 245-247.)

4.4 Inkrementaali rakentaminen

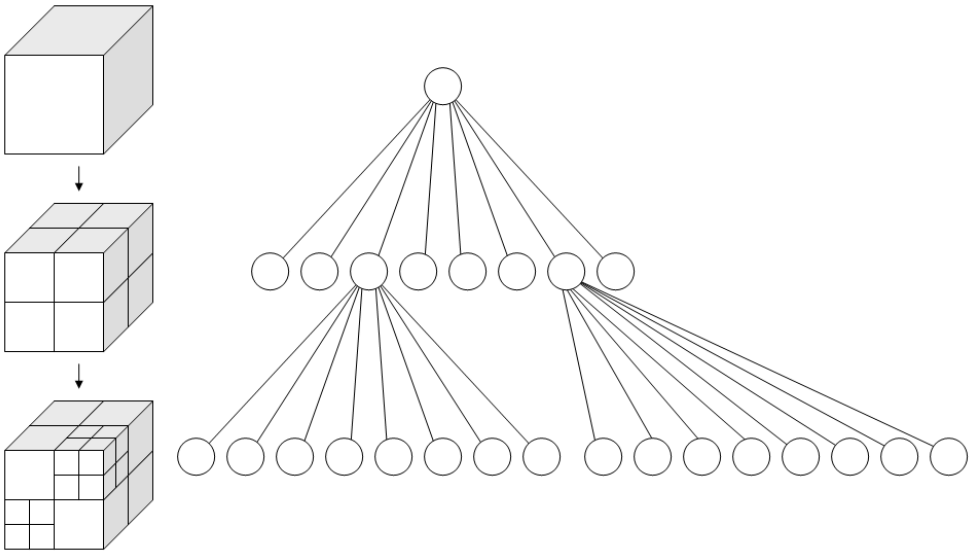
Hierarkia voidaan rakentaa myös syöttämällä yksi objekti kerrallaan, aloittaen tyhjästä puusta. Objektin paikka puussa valitaan jonkin kustannusfunktion avulla, joka arvioi objektin lisäämisen kustannuksen eri paikkoihin ja valitsee sitten halvimman (Andersen & Bay, 2006). Jos objekti on suuri verrattuna puussa jo oleviin objekteihin, se usein päätyy lähemmäs hierarkian latvaa. Pienemmät objektit todennäköisemmin kuuluvat jo olemassa oleviin rajauslaatikoihin, ja päätyvät lähemmäs hierarkian pohjaa. (Ericson 2005, 251.)

Koska puun rakenne riippuu siihen lisätyistä objekteista, ja koska objektien paikka puussa riippuu puun sen hetkisestä rakenteesta, on objektien syöttöjärjestyksellä merkitystä. Tämä on huono asia, koska samasta joukosta objekteja voidaan saada hyvin tasapainoinen ja tehokas hierarkia, tai huonoimmassa tapauksessa täysin epätasapainoinen ja huono hierarkia. (Andersen & Bay 2006, 5.)

4.5 Octree

Octree on eräs tunnetuimpia 3d-avaruuden jakamiseen tarkoitettuja tietorakenteita. Octree on puu-rakenne, jossa jokaisella solmulla on kahdeksan lapsisolmua. Jokaisella solmulla on pääakselien suuntainen (engl. axis-aligned) rajauslaatikko. Juurisolmun rajauslaatikko tyypillisesti pitää sisällään koko 3d-avaruuden geometrian niin tiiviisti kuin vain mahdollista. Juurisolmun rajauslaatikko on jaettu kahdeksaan pienempään, saman suuruiseen laatikkoon (kutsutaan myös oktanteiksi), halkaisemalla laatikko kerran jokaisella X, Y ja Z–pääakselilla. Nämä kahdeksan pienempää laatikkoa muodostavat juurisolmun lapsisolmut. (Ginsburg 2004, 439.)

Lapsisolmut ovat vuorostaan jaettu samalla tavalla, kukin kahdeksaan pienempään laatikkoon, jotka taas ovat jaettu vielä pienempiin laatikkoihin, ja niin edelleen (Kuvio 10.). Jakamista jatketaan rekursiivisesti, kunnes jokin kriiteri jakamisen lopettamiseksi täyttyy. Jakokertojen lukumäärä määrittelee myös octreen syvyyden. (Lengyel 2004, 237.)



Kuvio 10. Rajauslaatikon rekursiivinen jakaminen (WhiteTimberwolf).

4.5.1 Octree-solmun sisältämät tiedot

Octree luodaan yleensä esikäsittelyvaiheessa joko pelimoottorissa itsessään ennen varsinaista pelin alkua, tai jollain työkalulla pelin ulkopuolella. Octree-puun solmu koostuu vähintään seuraavista tiedoista:

- Rajauslaatikko
- Lista geometriasta
- Lista lapsisolmuista

Solmun sisältämä geometria voi periaattessa olla missä tahansa muodossa, mille vain voidaan tehdä törmäystarkistus rajauslaatikon kanssa, mutta yleensä reaaliaikaisissa ohjelmissa (kuten peleissä) on suotavaa käyttää kokonaisia 3d-objekteja. (Ginsburg 2004, 440.)

4.5.2 Lopetuskriteerit

Octree-solmujen jakaminen lopetetaan kun jokin seuraavista ehdoista täyttyy:

- Objektien tai polygonien minimimäärä solmussa on saavutettu, jolloin solmun jakamisesta ei enää olisi juurikaan hyötyä.
- Solmun rajauslaatikon mittasuhteet ovat liian pienet jakamista varten. (Periaatteessa kuutiot voidaan aina jakaa pienempiin osiin, mutta tietyn rajan jälkeen sitä ei välttämättä haluta tehdä.)
- Puun haluttu maksimisyvyys on saavutettu. Maksimisyvyydelle voidaan asettaa rajoituksia esim. muistin kulutuksen vuoksi. (Dunn & Parberry 2002, 395.)

Octreen (ja minkä tahansa puun) solmujen lukumäärä voidaan laskea kaavalla:

$$\frac{d^n - 1}{d-1} \qquad (5)$$

jossa *d* on solmusta haarautuvien lapsisolmujen määrä ja *n* on puun syvyys. Näin ollen seitsemän tasoisessa octreessa olisi jo lähes 300 000 solmua, käytännössä rajoittaen octreen syvyydeksi viisi tai kuusi tasoa. (Ericson 2005, 309.)

4.5.3 Octreen rakentaminen

Octreen rakentaminen on hyvin samanlainen prosessi kuin aikaisemmin kuvattu rajauslaatikkohierarkian ylhäältä-alas metodi. Alussa kaikki objektit ovat juurisolmussa. Algoritmin rakenne on seuraava:

- Objektit ovat solmussa P.
- Tarkistetaan halutaanko solmu vielä jakaa, eli täyttyykö mikään edellä mainituista kriteereistä ja jos ei täyty, niin jatketaan.
- Jaetaan P kahdeksaan tilavuudeltaan yhtäsuureen lapsisolmuun C1, C2, C3, ... , C8.
- Testataan objekti kerrallaan, onko se täysin lapsisolmun Cn –rajauslaatikon sisäpuolella. Jos on niin poistetaan se tästä solmusta ja lisätään lapsisolmun objektilistaan. Jos ei ole, niin toistetaan testi seuraavan lapsisolmun rajauslaatikon kanssa.
- Jos objekti ei ole täysin minkään lapsisolmun sisäpuolella (eli sen täytyy jäädä lapsisolmujen rajojen väliin), jätetään se tähän solmuun.
- Aloitetaan algoritmi alusta kaikille C1-C8 lapsisolmuista.

Octreeta rakennettaessa usein ilmaantuu tilanteita, jolloin pieni objekti osuu kahden solmun rajalle. Yleisin ja helpoin ratkaisu varmasti on ”korottaa” objekti ylempään solmuun. Tästä kuitenkin usein seuraa, että objekti kuuluu nyt tarpeettoman isoon rajauslaatikkoon ja on liian korkealla puuhierarkiassa, joka saattaa haitata octreen tehokkuutta.

Objekti on myös mahdollista halkaista kahteen pienempään objektiin, ja antaa halkaistu osio kummallekin solmulle. Paljon polygoneja sisältävän objektin halkaisu on kuitenkin hidasta ja vaikea toteuttaa.

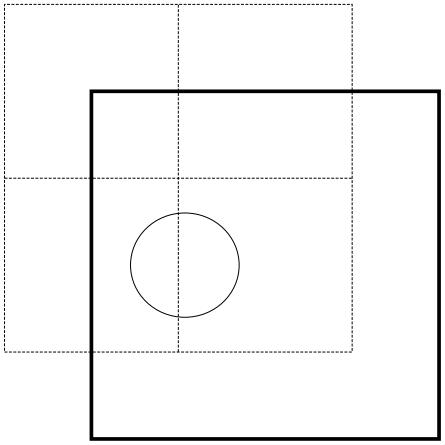
Eräs tehokas vaihtoehto on kasvattaa solmujen kokoa hieman niin, että solmut menevät osittain päällekkäin. Tälläistä kutsutaan ”loose octreeksi”.

4.5.4 Piirto octreen avulla

Näkyvien objektien piirto octreen avulla on yksinkertaista. Idea on tehdä leikkaustesti solmun rajauslaatikon ja kameran näkökartion kanssa. Jos rajauslaatikko on täysin kartion sisäpuolella, ovat myös kaikki lapsisolmujen rajauslaatikot kartion sisäpuolella, jolloin ne voidaan suoraan hyväksyä renderöitäväksi. Jos kartio leikkaa rajauslaatikkoa vain osittain, täytyy puuta seurata syvemmälle, ja toistaa leikkaustesti solmussa oleville objekteille sekä lapsisolmuille. Jos näkökartio ei leikkaa solmun rajauslaatikkoa ollenkaan, ei mikään objekteista tai lapsisolmujen objekteista tietenkään leikkaaa näkökartiota, eikä näin ollen ole myöskään näkyvissä. Octree-solmujen testaus näkökartiota vasten aloitetaan juurisolmusta.

4.6 Loose octree

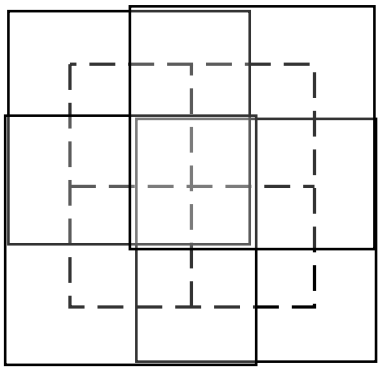
Vaikka perinteinen octree on tehokas ja suosittu tietorakenne, siinä on muutamia heikkouksia. Joissakin tilanteissa pieni objekti saattaa sijaita octreen solmujen rajalla (Kuvio 11.), jolloin se pienestä koostaan huolimatta joudutaan sijoittamaan korkealle puu-rakenteeseen, joka taas huonontaa octreehen suoritettavien hakujen tarkkuutta. (Ulrich 2000, 444.)



Kuvio 11. Objekti ei mahdu mihinkään tavalliseen octreen rajauslaatikoista, mutta se mahtuu loose octreen oikean alakulman laatikkoon. (Mukaillen Ulrich 2000, 449.)

Loose octree ratkaisee ongelman säätämällä solmujen rajauslaatikoiden kokoa niin, että naapurisolmujen rajauslaatikot menevät osittain päällekkäin. Tämä on esitetty kuviossa 12. Kat-

koviivalla kuvatut neliöt ovat normaaleja octreen rajauslaatikoita ja mustalla viivalla ovat samat rajauslaatikot loose octree mallisena. Laatikoita on piirretty hieman sivuun, jotta ne erottuisivat kuvassa paremmin. Solmujen keskikohta sekä puun rakenne pysyvät edelleen samoina kuin perinteisessä octreessä. (Ulrich 2000, 446.)

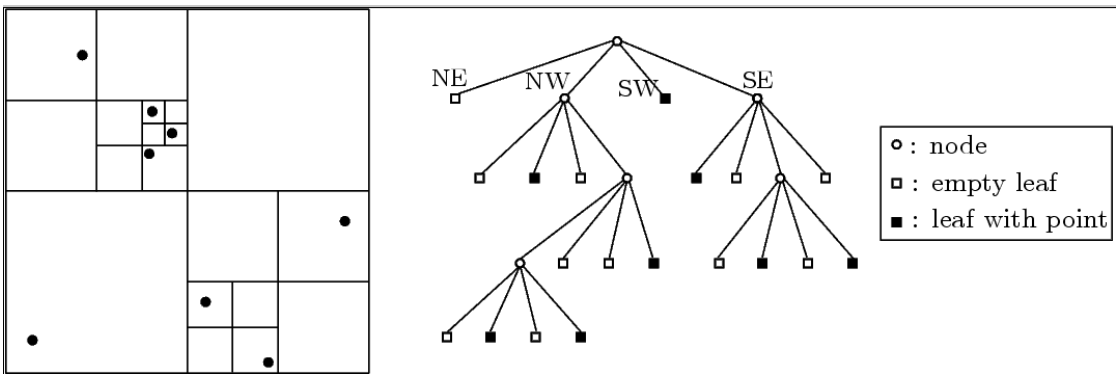


Kuvio 12. Octreen ja loose octreen rajauslaatikot (Mukaillen Ulrich 2000, 449).

Periaattessa rajauslaatikoiden kokoa ja päällekkäisyyttä voidaan säätää miksi tahansa, mutta yleensä suositellaan että rajauslaatikkoa kasvatetaan puolella mitalla joka suuntaan, jolloin koko rajauslaatikon tilavuus kasvaa kahdeksan kertaiseksi. (Ericson 2005, 318.)

4.7 Quadtree

Quadtree on octreen kaksiulotteinen versio. Siinä missä octreen solmuilla on kahdeksan haaraa/lapsisolmua, haarautuu quadtree neljään osaan, tai kvadranttiin. Octreen rajauslaatikko muoto on kuutio, kun taas quadtreessa se on neliö. Quadtreen rakentaminen aloitetaan ympäröimällä koko jaettava alue pääakselien mukaisella neliöllä. Tämä on juurisolmu. Tämä neliö jaetaan neljään yhtä suureen neliöön, jotka jaetaan taas neljään yhtä suureen neliöön (Kuvio 13.). Aivan kuten octreen luonnissa, tätä jakamista toistetaan rekursiivisesti kunnes tietyt kriiterit täyttyvät. Tätä kutsutaan ”region quadtreeksi”. Lapsisolmut voidaan nimetä ilmansuuntien mukaan NW, NE, SW ja SE. Tätä lapsisolmujen nimeämisjärjestystä kutsutaan nimellä Mortonin järjestys. (Puhakka 2008, 308.) Quadtreesta on kehitetty myös binääripuumainen versio (point quadtree), jolla voidaan esittää kaksiulotteista pistedataa (Puhakka 2008, 316).



Kuvio 13. Esimerkki quadtreesta. Vasemmalla aluen jako ruutuihin, ja oikeilla vastaava puurakenne (Samet 1984).

Quadtreeta hyödynnetään hyvin paljon peleissä maastografiikan piirtämisessä, jossa avaruutta tarvitsee jakaa vain kahdella akselilla. Myös quadtreesta voidaan muodostaa ”loose quadtree”, jossa lapsisolmut ovat osittain päällekkäisiä. Quadtreella on hyvin paljon samoja ominaisuuksia kuin octreella.

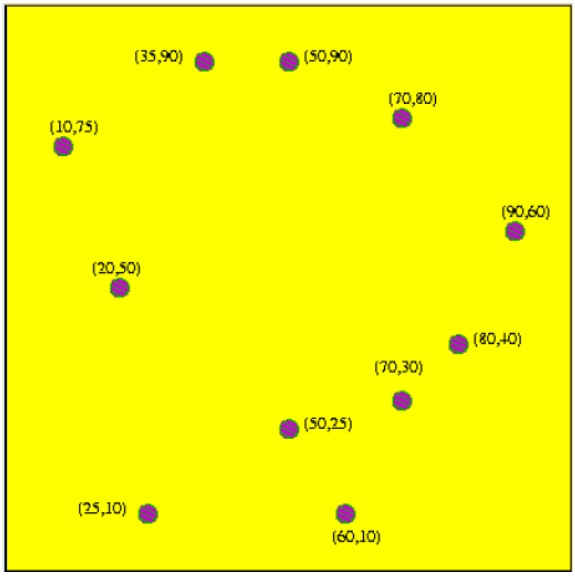
4.8 K-d-puu

Octree ja quadtree tyyppisten puiden laajentamisessa muihin ulottuvuuksiin on se ongelma, että kun avaruuden ulottuvuus on *k*, niin vastaavassa puussa on silloin 2^k haaraa. Tälläisessä puussa tyhjien solmujen viemä turha tila kasvaa. K-d-puu pyrkii osittain vastaamaan tähän ongelmaan. (Puhakka 2008, 315.)

Jon Louis Bentley esitteli *k*-dimensionaalisen binääripuun, eli k-d-puun, artikkelissaan (Bentley, 1975). Bentley tutki myöhemmin toisessa artikkelissaan (Bentley, 1980) k-d-puuta pistejoukon tallentamiseksi *k*-ulottuvuuksisesta avaruudesta. K-d-puu on BSP-puu (binary space-partitioning), jossa jakotasot ovat koordinaattiakselien suuntaiset. Tärkein ero on, että k-d-puussa tason paikka jolla avaruus jaetaan, ei ole kiinteä (Bikker, 2005). Tason tulee kuitenkin olla yhdensuuntainen jonkin pääakselin kanssa, ja jako suoritetaan vuorotellen jokaisella akselilla.

Jokaisessa puun rakennusvaiheessa valitaan yksi koordinaattiakseli lähtökohdaksi jäljellä olevien pisteiden jakamiseksi. Usein jako suoritetaan toistuvasti vuorotellen jokaisella pääakselil-

la (ensin x-, sitten y- ja z-akselilla), mutta tätä toistuvuutta ei ole pakko noudattaa ja akseliksi voi valita minkä tahansa k-ulottuvuuksista. Esimerkkinä toteutetaan Bentleyn kuvaama k-d-puu kaksiulotteiselle pistejoukolle (Kuvio 14.). Pisteet voidaan syöttää yksitellen tai kaikki kerrallaan.



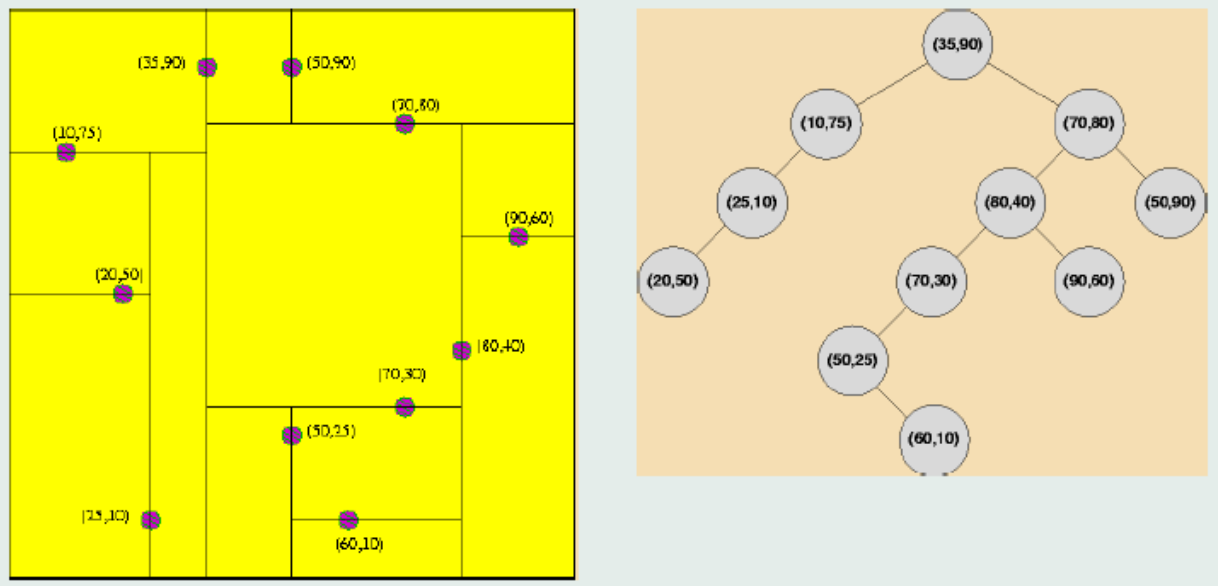
Kuvio 14. Pistejoukko x- ja y-koordinaatteineen (Chandran 2002).

Ensin valitaan juurisolmulle aloituskohta ja jaetaan avaruus x-akselin mukaan. Tästä seuraa että:

- Kuten kaikissa binääripuissa, kaikilla juurisolmun vasempaan haaraan jäävillä/tallennattavilla pisteillä on pienempi x-koordinaattiarvo kuin juurisolmulla.

- Kaikilla juurisolmun oikeinpuoleiseen haaraan jäävillä pisteillä on suurempi tai yhtä suuri x-akselin koordinaattiarvo kuin juurisolmulla.

Seuraavaksi jaetaan molemmat juurisolmun lapsisolmuista jostakin kohtaa y-akselin mukaan. Tästä syntyvät yhteensä neljä uutta lapsisolmua taas jaetaan x-akselin mukaan, joiden lapsisolmut y-akselin mukaan, ja niin edelleen.



Kuvio 15. Vasemmalla jaettu pistejoukko, ja oikealla vastaava puurakenne (Chandran 2002).

Staattisen k-d-puun rakentaminen n –määrästä pisteitä voidaan suorittaa O(*n* log^2 *n*) ajassa. K-d-puuta hyödynnetään paljon myös säteenjäljityksessä (Bikker, 2005).

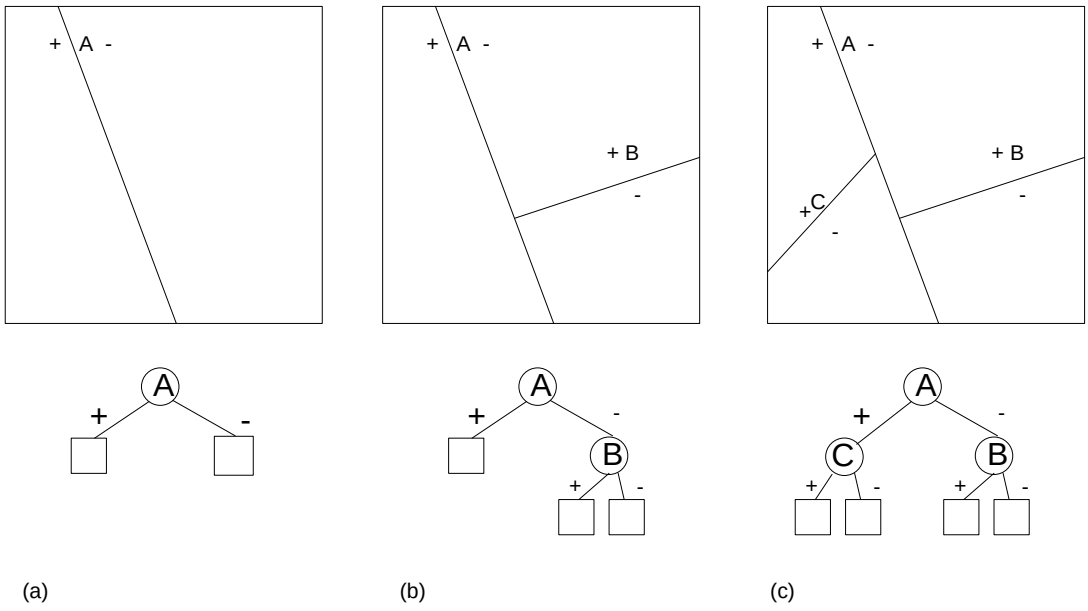
4.9 BSP-puu

BSP-puut esiteltiin ensimmäisen kerran Schumacher ja kump. toimesta jo vuonna 1969. BSP-puun tarkoituksena oli tarjota ratkaisu piilossa olevien pintojen etsimiseen ja poistamiseen piirrettävästä ympäristössä. Suosituksi peleissä BSP-puu tuli 1990-luvun alussa Id Softwaren ohjelmoija John Carmackin toimesta, koska hänen Doom-pelinsä käytti sitä hyvällä menestyksellä. (Abrash, 2001.) Vanhoissa 3d-peleissä BSP-puu oli tehokas tapa piirtää polygonit oikeassa järjestyksessä taaimmaisesti etummaiseen, mutta nykyään 3d-näytönohjaimet hoitavat tämän automaattisesti ja tehokkaammin laitteistokiihdytettynä Z-puskurin avulla. (Sherrod 2008, 544.)

Spatiaalisista tietorakenteista BSP-puu on kaikista joustavin. Sillä pystyy matkimaan muita tietorakenteita kuten k-d-puuta, quad- ja octreetä, mutta muut tietorakenteet eivät pysty kaikkeen mihin BSP-puu. BSP-puu soveltuu myös erinomaisesti törmäystarkistuksiin. (Ericson 2005, 349.)

BSP (binary space-partitioning) puu on binääripuu, joka jakaa avaruuden rekursiivisesti kahteen aliavaruuteen. Käytännössä jos kyseessä on 3d-avaruus, niin jakaminen suoritetaan tasolla, jos taas 2d-avaruus, niin jakaminen suoritetaan viivalla. Jakamisessa käytettävää tasoa kutsutaan jakotasoksi (engl. splitting plane). Kahta puoliavaruutta (engl. halfspace), joksi avaruus jakautuu, kutsutaan positiiviseksi ja negatiiviseksi puoliavaruudeksi. Positiivinen puoliavaruus sijaitsee tason edessä ja negatiivinen tason takana. Toisin kuten esim. octreessa, BSP-puussa jakotasojen ei tarvitse olla pääakselien suuntainen, vaan taso voi olla suunnattu miten tahansa. (Ericson 2005, 350.)

Avaruuden jakamista toistetaan rekursiivisesti. Jokaisesta jakokerrasta puu-rakenteeseen lisätään uusi solmu, joka tallentaa tason jonka mukaan solmu on jaettu, listan polygoneista tai objekteista sekä osoittimet lapsisolmuihin. Prosessi on havainnoillistettu kuviossa 16.



Kuvio 16. Tilan jakaminen tasolla, ja vastaava puurakenne (Mukaillen Ericson 2005, 350). (a) ensimmäinen jako ja juurisolmu. b) ensimmäinen toisen-tason jako. c) toinen toisen-tason jako.

BSP-puut voidaan jakaa kolmeen tyyppiin sen mukaan tallentavatko ne dataa sisäisiin solmuihin (solmukohtainen, engl. node-based bsp) vai lehtisolmuihin (lehtikohtainen, engl. leaf-based tai leafy bsp). Kolmantena tyyppinä voidaan pitää lehtikohtaisen bsp:n erikoistapausta, niin kutsuttua ”solid-leaf” bsp:tä.

Solmukohtainen on perinteinen BSP-puu, jossa jakotasona toimii yksi polygoneista ja geometria sekä jakotasot sijaitsevat puun sisäisissä solmuissa. Lehtisolmut ovat tyhjiä ja avaruuden jakamista jatketaan kunnes kaikki polygonit on käyty läpi. Lehtikohtaisessa puussa sisäiset solmut sisältävät ainostaan jakotasot ja osoittimet lapsisolmuihin. Geometria tallennetaan puun lehtisolmuihin. (Walsh 2008, 189.)

4.9.1 BSP-puun rakentaminen ja jakotason valinta

BSP-puun rakentamisen vaiheet ovat yksinkertaiset:

- Valitse jakotaso.
- Jaa geometria jakotasolla kahteen ryhmään (jakotason negatiivisella puolella olevaan ja jakotason positiivisella puolella olevaan geometriaan).
- Toista rekursiivisesti molemmille ryhmille.

Rekursion lopettaminen riippuu BSP-puun tyypistä sekä tarkoituksesta, johon puuta rakennetaan. Tyypillisiä kriteerejä ovat solmun polygonien maksimäärän tai puun syvyyden saavuttaminen (BSP FAQ).

Puuta luodessa täytyy myös päättää haluaako tasapainoisen puun, jossa solmun vasemman ja oikeanpuolen alipuiden syvyydessä ei ole liian isoa eroa. Kuten muissakin puurakenteissa, tasapainoiseen puuhun tehdyt haut ovat nopeampia. (Ranta-Eskola, 2003.)

Puun ensimmäinen jakotaso (eli juurisolmun jakotaso) vaikuttaa suoraan puun kokoon. Jos jakotaso leikkaa liian monia polygoneja, puusta tulee luultavasti liian iso. Jakotasoksi halutaan sellainen taso, jolla minimioidaan halkaistavien polygonien määrä, ja mikä pitää puun mahdollisemman tasapainoisena. (Shimer 1997.)

Täysin optimaalisen ja parhaan BSP-puun rakentamista varten täytyisi jakotasoksi kokeilla jokaista polygonia jokaisella tasolla. Tämän aikakompleksisuus on O(n!), joten tälläinen ”brute force” etsiminen on hyvin epäkäytännöllistä pienilläkin BSP-puilla.

Helppo tapa jakotason valintaan on testata muutamaa, satunnaisesti valittua polygonia jakotasona, ja valita niistä paras. Näin ainakin vältytään valitsemasta huonointa mahdollista vaihtoehtoa. (Dunn & Parberry 2002, 401.)

4.9.2 Objektikohtainen BSP-puu

Koska 3d-peleissä polygonien määrä on noussut rajusti, ja yhdessä 3d-ympäristössä saattaa olla jopa kymmeniä miljoonia polygoneja, ei niitä kaikkia voida käsitellä yksitellen. Tämä valitettavasti tekee yksittäisiä polygoneja käsittelevistä solmukohtaisista BSP-puista vanhanaikaisia ja käyttökelvottomia nykypäivän tarpeisiin. Tehokkaasti toimiakseen nykyaikaiset laitteistokiihdytetyt 3d-näytönohjaimet vaativat, että niille lähetetään yksittäisten kolmioiden sijasta isompia ryhmiä polygoneja kerrallaan. Järkevämpää onkin tehdä jakaminen objekti-tasolla, sekä valita jakotasot jollakin muulla perusteella. Staattiset objektit joita jakotaso leikkaa, voidaan jakaan kahdeksi pienemmäksi objektiksi tai korottaa korkeammalle BSP:n tasolle. Tälläisellä objekti-tason BSP-puulla saavutetaan sama logaritminen nopeusetu, samalla kuitenkin lähettäen näytönohjaimelle kokonaisia objekteja piirrettäväksi.

5 OCTREE LISÄOSA KAJAK3D PELIMOOTTORIIN

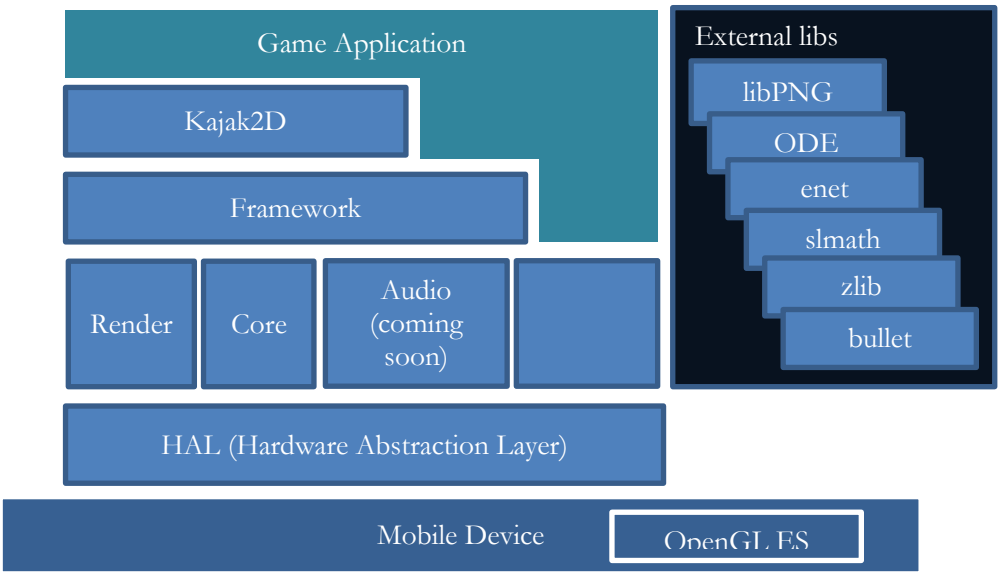
Tämä opinnäytetyön tavoitteena oli suunnitella ja toteuttaa yleiskäyttöinen näkyvyystarkastelulisäosa Kajaanin Ammattikorkeakoulun kehittämään Kajak3D-mobiilipelimoottoriin. Kajak3D pelimoottorissa ei vielä ole minkäänlaista näkyvyystarkastelu tai skenenhallintaa, joten lisäosa pyrkii vastaamaan tähän tarpeeseen.

5.1 Kajak3D pelimoottori

Kajak3D on Kajaanin Ammattikorkeakoulun kehittämä mobiililaitteille suunnattu pelimoottori. Kajak3D on toteutettu C++ ohjelmointikielellä Nokian kehittämän JSR-184 ja JSR-297 mobiiligrafiikkaspesifikaatioiden mukaan. Kaikki malli- ja tekstuuriresurssit ovat JSR:n määrittelemässä M3G –tiedostomuodossa, joka mahdollistaa kokonaisten ympäristöjen tuomisen 3d-mallinnusohjelmasta Kajak3D:hen. Tätä kirjoittaessa Kajak3D tukee seuraavia laiteympäristöjä:

- Windows PC (OpenGL)
- iPhone (OpenGL ES 1.x)
- Nokia N900/Maemo (OpenGL ES 1.x ja 2.0)
- Android (OpenGL ES 1.x ja 2.0)
- Samsung Bada (OpenGL ES 1.x ja 2.0)

Vaikka toistaiseksi Kajak3D:tä käytetään pääasiassa Kajaanin Ammattikorkeakoulun pelialan opetuksessa, on pelimoottori vapaasti myös muiden oppilaitosten ja kainuulaisten peli-alan yritysten käytettävissä. Käyttäjän ei tarvitse osata monimutkaisia OpenGL ja DirectX grafiikkarajapintoja, vaan hän toteuttaa sovelluksensa Kajak3D:n rajapinnan avulla, jolloin sovellus toimii automaattisesti kaikilla tuetuilla alustoilla. Pelimoottori on toteutettu hyvin joustavasti, ja se on mahdollista laajentaa ylöspäin PC ja konsoliympäristöihin. Tulevaisuudessa pelimoottorin kehityksen painopiste on apu- ja työkaluohjelmissa, kuten kenttä- ja partikkelieditorissa.



Kuvio 17. Kajak3D arkkitehtuuri kaavio

Kuviossa 17. on esitelty Kajak3D:n arkkitehtuuri kirjastotasolla. Alimmaisena on laitteistotaso, jonka pelimoottori abstrahoi pois HAL (Hardware Abstraction Layer) tason avulla. Tämän päälle on rakennettu muut ydinkirjastot, kuten *Render* ja *Core*. Samalla abstraktiotasolle sijoittuu myös *Scenemanagers* –lisäosa, joihin siis octree-lisäosakin kuuluu. Lisäosalla ei ole riippuvaisuuksia *Framework* –tasolle, mutta se on kyllä riippuvainen *Render* ja *Core* -moduleista.

5.2 Vaatimusmäärittely

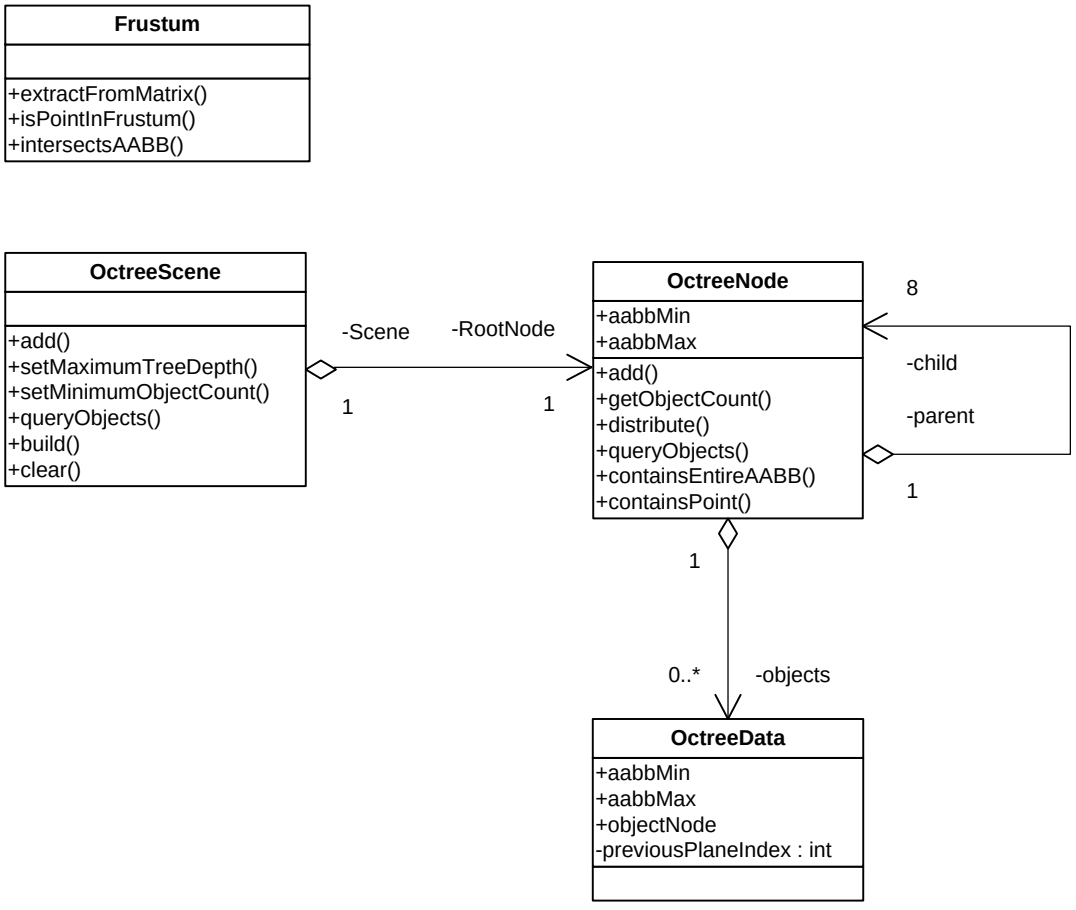
Octree-lisäosan vaatimuksista sovittiin yhdessä Kajaanin Ammattikorkeakoulun pelilaboratorion henkilökunnan kanssa. Koska Kajak3D on pääasiassa älypuhelimille suunnattu pelimoottori, tulisi lisäosan toimia kaikilla tuetuilla mobiilialustoilla. Aluksi todettiin, että lisäosaa käytetään ennenkaikkea staattisten objektien näkyvyystarkasteluun, joten lisäosa voidaan optimoida erityisesti tähän tarkoitukseen. Dynaamisten objektien tuki jätetään pois, ja se voidaan toteuttaa jollain paremmin niille sopivammalla tietorakenteella myöhemmin.

Octree-lisäosassa tulisi olla runsaasti konfiguraatioparametreja, jolloin sitä pystyttäisiin räätälöimään jokaiseen käyttötapaukseen erikseen. Octreen käyttö ei myöskään saisi tarpeettomasti rajoittua pelkästään piirtoon, vaan siihen pitää pystyä tekemään myös yleisiä kyselyjä objek-

tien paikoista, jolloin sitä voidaan hyödyntää esimerkiksi myös törmäystarkistuksissa tai verkkopeli-koodissa. Toteutuksen tulisi tietenkin olla tehokas ja muistia säästävä.

5.3 Toteutus

Koska Kajak3D:n toteutuskieli on C++, luonnollisesti myös octree-lisäosa toteutettiin sillä. Kajak3D:hen luotiin uusi nimiavaruus *Scenemanager*, jonka alle on tarkoitus toteuttaa erilaisia skenenhallinta- ja näkyvyydentarkastelualgoritmeja. Octree on näistä ensimmäinen. Kuten hyviin ohjelmointikäytäntöihin kuuluu, toteutus haluttiin pitää mahdollisimman yksinkertaisena ja riippuvuudet minimaalisena. Koko lisäosa koostuukin vain neljästä luokasta, joista kolme on octree-luokkia, ja yksi on apuluokka näkökartion leikkaustestejä varten. Lisäosan luokkarakenne on esitelty kuviossa 18.



Kuvio 18. Octree-toteutuksen luokkakaavio.

Frustum –luokka on kameran näkökartion ja objektien leikkaustestejä varten. Luokalle annetaan *extractFromMatrix()* –metodin parametrina kameran projektio-matriisi, josta se purkaa näkökartion kuusi tasoa. Luokan avulla voidaan testata onko esimerkiksi tietty piste tai laatikko näkökartion sisäpuolella.

OctreeScene tarjoaa varsinaisen rajapinnan, jonka avulla octree-lisäosaa käytetään. OctreeScene hallinnoi koko octree-rakennetta, sekä siihen syötettäviä objekteja ja tehtyjä kyselyitä. Koska kaikilla octreehen tallennettavilla objekteilla tulee olla rajauslaatikko, voi OctreeSceneen syöttää vain *render::node* –tyyppisiä osoittimia. Kaikki Kajak3D:n geometriset objektit kuten mallit, spritet ja partikkelit periyttävät *render::node* –rajapinnan, jossa on metodit rajauslaatikkotietojen kysymistä varten.

Varsinainen octree-hierarkian toteutus on OctreeNode –luokassa, joka on toteutettu niin kutsutulla kompositio –suunnittelumallilla (Geary, 2002). OctreeNode sisältää tiedon solmun rajauslaatikosta, sekä osoittimet kahdeksaan lapsisolmuun. Octreen hakujen toiminallisuus ja objektien lisäämisen ovat tässä luokassa.

Käyttäjän syöttämät nodet ovat kääritty OctreeData -luokkaan, jossa render::node –osoittimen lisäksi ovat tiedot noden rajauslaatikosta, sekä tietoa edellisessä framessa tehdystä leikkaustestistä, jota käytetään hakujen optimoimiseen.

Octreeta voidaan konfiguroida kolmella parametrilla, jotka ovat:

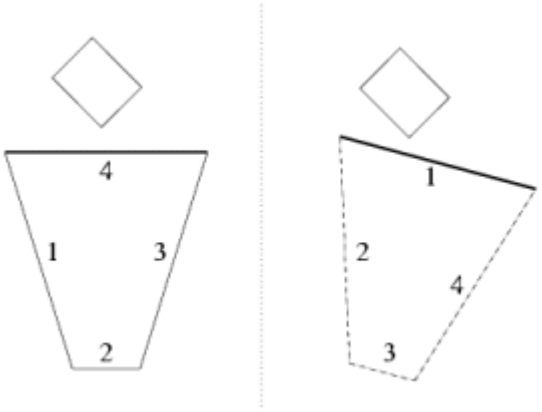
- Puun maksimisyvyys. Tällä parametrilla voidaan estää puun solmujen lukumäärän kasvua liian suureksi. Parametrin sallittu minimiarvo on 1. Maksimiarvoa ei ole, mutta useimpiin käyttötapauksiin 3 tai 4 on hyvä arvo.

- Solmun objektien minimimäärä. Tällä parametrilla määritellään kuinka monta objektia solmussa on oltava, ennen kuin se jaetaan lapsisolmuiksi.

- Solmun kolmioiden minimimäärä. Tällä parametrilla voidaan asettaa solmun jakamiseen tarvitta objektien yhteenlaskettujen kolmioiden minimimäärä. Solmun jakamista ei kannata tehdä, mikäli solmussa olevissa objekteissa on vain vähän kolmioita.

Normaalia octreen ja näkökartion tarkistusta on mahdollista nopeuttaa ottamalla huomioon edellisen tarkistuksen tulokset. Jos edellisessä kuvassa (framessa) jokin rajauslaatikko ei lei-

kannut kameran näkökartiota, niin se luultavasti ei leikkaa sitä myös seuraavassakaan kuvassa. Tätä optimointia varten OctreeData –luokkaan tallennetaan näkökartion tason indeksi, johon leikkaustarkistus edellisessä kuvassa epäonnistui ja seuraavassa kuvassa leikkaustarkistus aloitetaan tästä tasolla. Näin rajauslaatikko voidaan hylätä jo aikaisessa vaiheessa. Tasojen testausjärjestyksen vaihtoa on havainnollistettu kuviossa 19.



Kuvio 19. Näkökartion tasojen uudelleen järjestely.

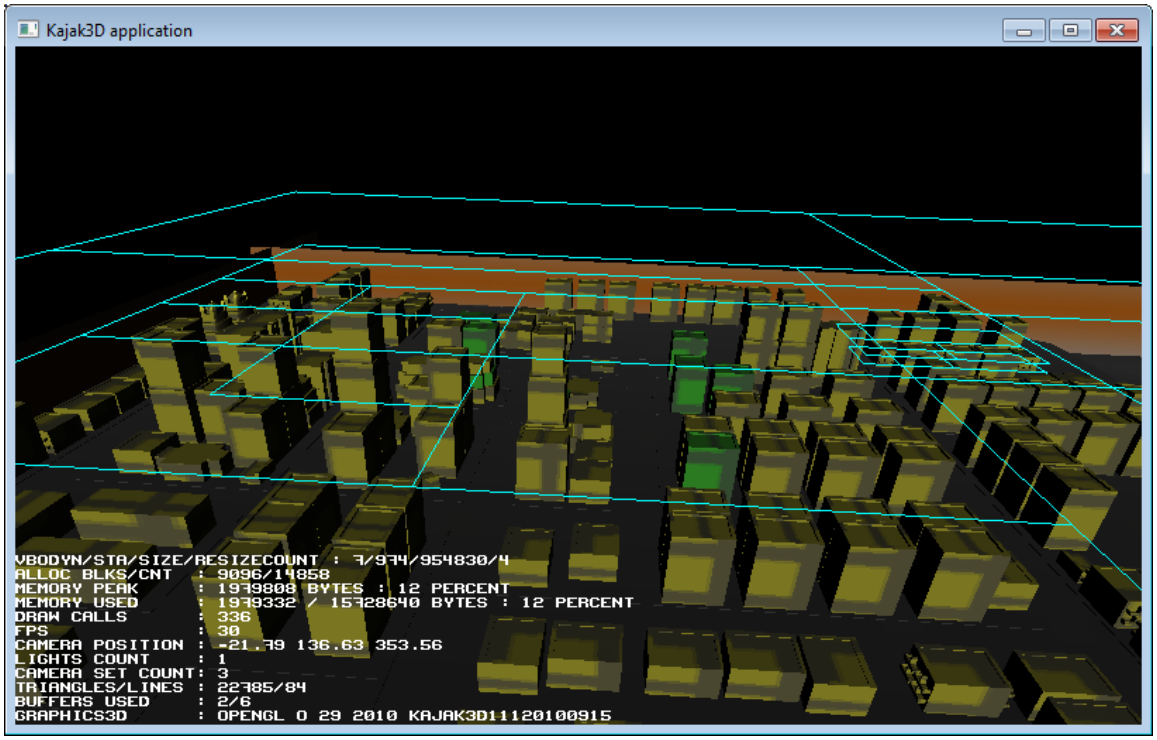
Toinen toteutettu optimointi liittyy tason ja rajauslaatikoiden leikkaustarkistukseen. Kaikkien laatikon kulmapisteiden testaamisen sijaan, rajauslaatikko voidaan hyväksyä tai hylätä maksimissaan kahden pisteen tarkistuksella. Nämä pisteet, joita kutsutaan n- ja p-vertekseiksi (negatiivinen- ja positiivinen takapiste), saadaan ottamalla kauimmainen piste tason normaalin positiiviselta sekä negatiiviselta puolelta. Ulf Assarsson ja Tomas Möller kuvailivat nämä ja muutamia muita näkökartion optimointeja paperissaan (Assarsson & Möller, 1999).

5.4 Testaus ja testitulokset

Tuotetun octree-lisäosan tehokkuutta voidaan arviodia kahdella tavalla. Octreen kautta tehtyjä leikkaustestien lukumäärää voidaan verrata perusnäkökartion näkyvyystarkistukseen, eli siihen kun näkökartion leikkausta testataan yksi kerrallaan jokaisen objektin kanssa. Toinen vertailukohde on suora tehonlisäys, eli paljonko sekä näkyvyystarkistus että ympäristön piir-

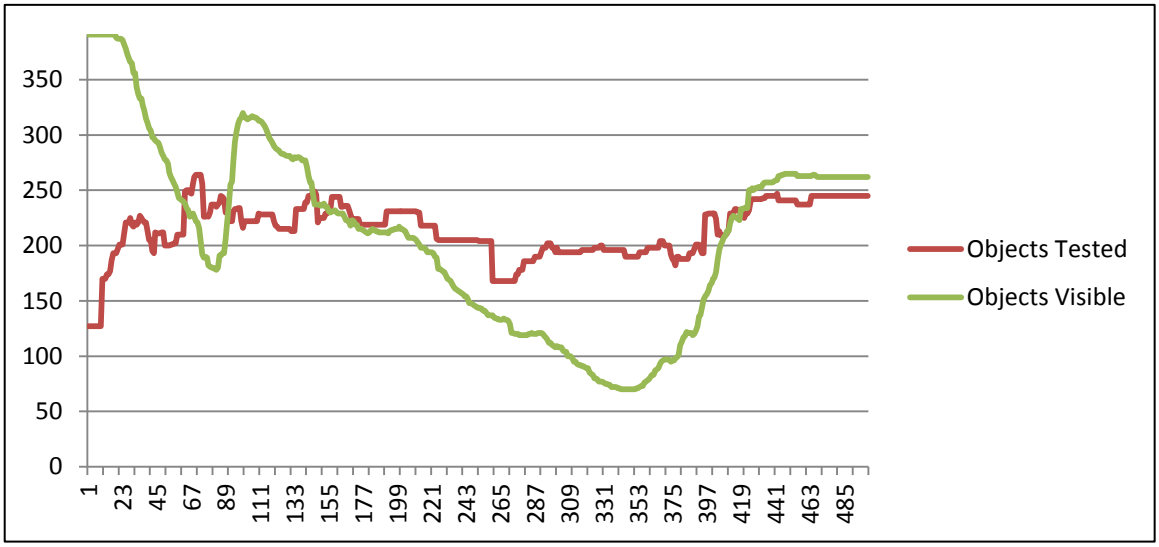
täminen nopeutuivat verrattua perustarkistuksiin, ja siihen että koko ympäristö mukaan lukien piilossa olevat objektit piirrettäisiin.

Testaus suoritettiin tekemällä kaksi testiä 500 ruudun pituinen kamera-ajo virtuaalisessa kaupunkiympäristössä (Kuvio 20.) esiohjelmoitua reittiä pitkin. Kaupunkiympäristössä oli 391 objektia. Ensimmäisessä testissä tutkittiin, paljonko octree vähentää leikkaustestejä verrattuna tavalliseen lineaariseen näkökartion leikkaustestaukseen, jossa jokaista objektia testataan näkökartiota vasten. Testissä jokaisella ruudulla kirjattiin ylös näkyvien objektien ja tehtyjen leikkaustestien lukumäärä.



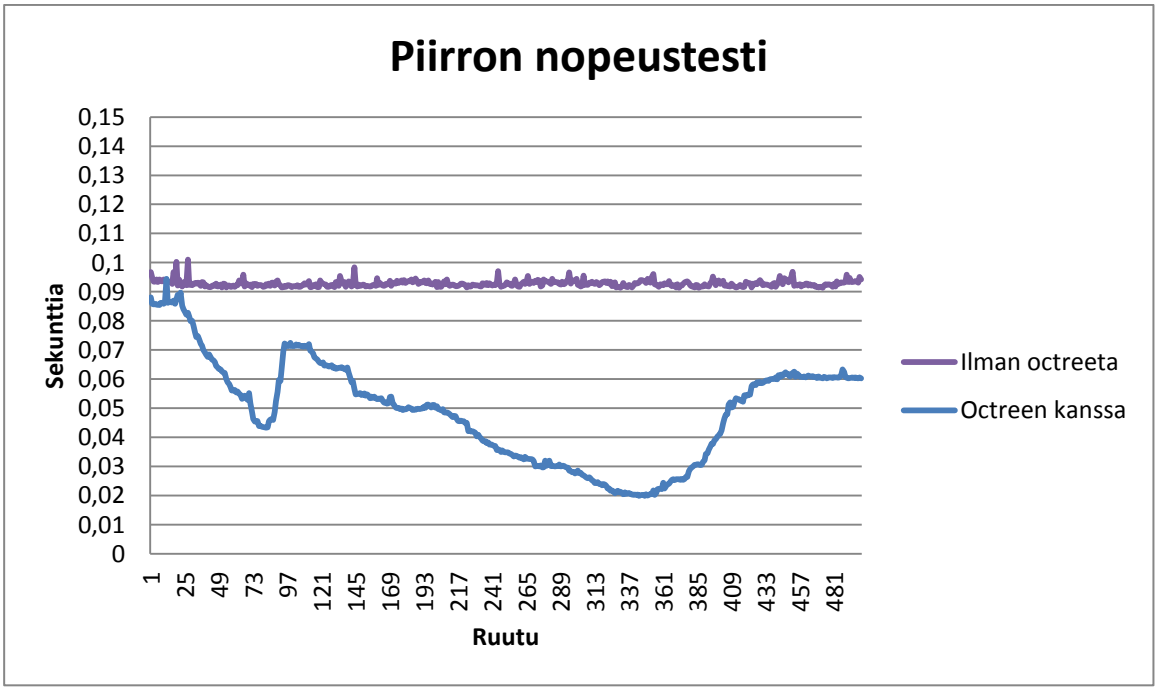
Kuvio 20. Testeissä käytetty kaupunki-ympäristö. Objekteja sisältävät octree-solmujen rajat näkyvät vaaleina viivoina.

Testin alussa kamera kuvaa kaupunkia niin, että kaikki objektit ovat samaan aikaan näkyvissä. Kuviosta 21. voidaan havaita että juuri silloin objektikohtaisia leikkaustestejä tehdään vähiten, koska yleensä suuria alueita saadaan hyväksyttyä suoraan ilman yksittäisiä testejä.



Kuvio 21. Näkyvien objektien ja testattujen objektien määrät kuvittain

Kameran lähestyessä kaupunkia, lopulta lentäen talojen välistä, näkyvien objektien määrä pienenee ja leikkaustestien määrä kasvaa, koska näkökartiota osittain leikkaavia octree-solmuja on yhä enemmän. Toisella testillä mitattiin ruudun päivitys- ja piirtonopeutta octreen kanssa ja ilman. Testin tulokset ovat esitelty kuviossa 22.



Kuvio 22. Ruudun päivitys- ja piirtoaika ruuduittain. Kulunut aika on sekunneissa.

Jokaisella ruudulla ajastettiin ruudunpäivitysfunktioon, sekä piirtofunktioon kuluva aika. Ruudunpäivitysfunktiossa kysyttiin octreelta kaikki kameran näkökartiota leikkaavat objektit, jotka sen jälkeen piirrettiin piirtofunktiossa. Ilman octreeta päivitysfunktio oli tyhjä, ja piirtofunktiossa piirrettiin koko ympäristö. Kuvion 22. tuloksista nähdään kuinka octreen kanssa ruudun piirtoon kulunut aika vaihtelee paljon, ja jos sitä verrataan kuvion 21. näkyvien objektien käyrään, huomataan vahva korrelaatio näkyvissä olevien objektien määrän ja piirtoon kuluneen ajan välillä. Ilman octreeta piirtoon kulunut aika on loogisesti jokaisella ruudulla lähes sama, lukuunottamatta pientä kohinaa. Tuloksista ilmenee selvästi, että piirto octreen kanssa oli huomattavasti nopeampaa kuin ilman octreeta. Parhaimmillaan nopeuseroa octreen hyväksi oli lähes 80 prosenttia. Molemmat testit suoritettiin Android 2.1 -käyttöjärjestelmällä Motorola Milestone puhelimessa, käyttäen Kajak3D:n OpenGL ES 1.1 renderöijää.

5.5 Jatkokehitysmahdollisuudet

Lisäosan jatkokehitys mahdollisuuksia on runsaasti. Erityisesti dynaamisten, eli liikkuvien objektien tuki oli pitkään harkinnassa jo nykyiseen lisäosaan, mutta ajanpuutteen vuoksi siitä päätettiin luopua. Toinen hyvä kehitysmahdollisuus on jonkinlaisen peittävyystarkastelualgoritmin lisääminen. Haasteita molempiin asettaa toistaiseksi mobiililaitteiden heikko laskentateho verrattuna PC-laitteisiin, mutta niissäkin kehitys on todella nopeaa ja tämä on varmasti vain tilapäinen este monimutkaisempien näkyvyystarkasteluiden toteuttamiselle mobiilialustoilla. Jatkokehitystä varten saattaa olla hyödyllistä testata ja arvioida myös muita tietorakenteita ja harkita nykyisen octreen korvaamista jollakin toisella. Toinen vaihtoehto on tehdä niistä kokonaan omat lisäosansa scenemanager –kirjastoon.

6 POHDINTA

Opinnäytetyön idea tuli ohjaavalta opettajalta Veli-Pekka Piiraiselta. Aihe tuntui mielenkiintoiselta alusta lähtien, mutta epäilin vahvasti olisiko se riittävän laaja kokonaiseksi opinnäytetyöksi. Tämä osoittautui hyvin pian harhaluuloksi, sillä aiheena näkyvyyden ongelma ja näkyvyystarkastelu on valtava. Akateemisia tutkimuksia ja papereita aiheesta on julkaistu satoja, ellei tuhansia. Toisaalta taas alan kirjallisuus on harmittavan yksipuolista ja koostuu lähinnä yleisistä peliohjelmointikirjoista, joista jokainen selittää enemmän tai vähemmän samat asiat hieman eri sanoin. Tämän opinnäytetyön tarkoitus on tutustuttaa lukija yleisimpiin näkyyvystarkastelussa käytettyihin tietorakenteisiin ja tekniikoihin, sekä tarjota riittävästi tietoa hierarkisista algoritmeista että lukija voi niiden pohjalta myös toteuttaa esimerkiksi octree tai rajauslaatikkohierarkia tietorakenteen. Mielestäni opinnäytetyö onnistuu tässä kohtuullisesti. Hierarkisia rakenteita selittävä osio on kattava, mutta peittävyystarkasteluosio jäi ajan puutteen vuoksi harmittavan pinnalliseksi.

Käytännöntyöstä, eli octree-toteutuksesta tuli toimiva ja hyödyllinen, mutta valitettavasti vain minimitavoitteet saatiin toteutettua ajanpuutteen vuoksi. Työn ideointi ja vaatimusmäärittely on tehty lähinnä Jani Kajalan kanssa käytyjen keskustelujen pohjalta. Jani Kajala suoritti myös pikaisen katselmoinnin ja laaduntarkistuksen octree-lisäosan lähdekoodille. Mikäli aikaa olisi jäänyt enemmän, olisi työhön voinut tutkia dynaamisten objektien tallennusta sekä mobiililaitteistolle sopivia peittävyydentarkistustekniikoita. Myös muita tietorakenteita ja niiden välisiä nopeus- ja soveltuvuuseroja olisi ollut mielenkiintoista kokeilla. Käytännöntyötä testattaessa viimein selvisi myös näkyvyysoptimoinnin todellinen tarpeellisuus, sillä hyöty mikä sen avulla voidaan saavuttaa on valtava. Olen vakuuttunut, että näkyvyysoptimointi on varmasti tärkein ruudunpäivitys nopeuteen vaikuttava tekijä missä tahansa nykyaikaisessa laajoja ympäristöjä sisältävässä 3d-pelissä. Mielestäni tätä ei painoteta tarpeeksi peliohjelmointia käsittelevissä kirjoissa eikä oppitunneilla.

LÄHTEET

Akenine-Moller, T., Haines, E. & Hoffman, N. 2008. Real-Time Rendering, Third Edition. Wellesley: AK Peters.

Carter, B. 2004. Game Asset Pipeline. Boston: Charles River Media.

Dunn, F & Parberry, I. 2002. 3D Math Primer for Graphics and Game Development. Texas: Wordware Publishing Inc.

Ericson, C. 2005. Real-time Collision Detection.San Francisco: Elsevier Inc.

Ginsburg, D. 2004. Octree Construction. Game Programming Gems: Charles River Media.

Gregory, J. 2009. Game Engine Architecture. Wellesley: A K Peters.

Lengyel, E. 2004. Mathematics for 3D Game Programming and Computer Graphics Second Edition. Massachusetts: Charles River Media, Inc.

Puhakka, A. 2008. 3D-Grafiikka.Helsinki: Talentum.

Sherrod, A. 2008. Game Graphics Programming. Boston: Course Technology.

Ulrich, T. 2000. Loose Octrees. Game Programming Gems. Boston: Charles River Media.

Walsh, P. 2008. Advanced 3D Game Programming With DirectX 10.0. Texas: Wordware Publishing Inc.

WEB-LÄHTEET

Abrash, M. 2001. Graphics Programming Black Book. Saatavilla www-muodossa http://www.phatcode.net/res/224/files/html/ch60/60-01.html (Luettu 1.8.2010)

Andersen, K. & Bay, C. 2006. A survey of algorithms for construction of optimal Heterogeneous Bounding Volume Hierarchies. Saatavilla pdf-muodossa http://image.diku.dk/projects/media/christian.bay.kasper.andersen.06B.pdf (Luettu 6.7.2010)

Assarsson, Ulf & Möller, T. 1999. Optimized View Frustum Culling Algorithms. Technical Report 99-3. Chalmers University of Technology. Department of Computer Engineering. Saatavilla pdf-muodossa http://www.cse.chalmers.se/~uffe/vfc.pdf (Luettu 29.9.2010)

Bentley, J. 1974. Multidimensional Binary Search Trees Used for Associative Searching. Saatavilla pdf-muodossa www.cs.utk.edu/~dbanks/courses/2010.0/assignment/kdtree-bentley.pdf (Luettu 4.9.2010)

Bentley, J. 1980. Multidimensional Divide-and-Conquer. Saatavilla pdf-muodossa http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.8849&rep=rep1&type=pdf (Luettu 4.9.2010)

Bikker, J. 2005. Raytracing: Theory & Implementation Part 7, Kd-Trees and More Speed. Saatavilla www-muodossa http://www.devmaster.net/articles/raytracing_series/part7.php (Luettu 12.8.2010)

BSP FAQ. BSP Tree Frequently Asked Questions. Saatavilla ftp-osoitteessa ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html (Luettu 2.8.2010)

Chandran, S. Introduction to kd-trees. Saatavilla pdf-muodossa http://www.cs.umd.edu/class/spring2002/cmsc420-0401/pbasic.pdf (Luettu 14.8.2010)

Fernandes, A. R. View Frustum Culling Tutorial. Saatavilla www-muodossa http://www.lighthouse3d.com/opengl/viewfrustum/ (Luettu 27.8.2010).

Geary, D. 2002. A look at the Composite design pattern. Saatavilla www-muodossa http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html (Luettu 10.9.2010)

Gottschalk, S., Ming, L. & Dinesh, M. 1996. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. Saatavilla pdf-muodossa http://www.cs.unc.edu/~walk/papers/gottscha/sig96.pdf

Portal Culling. 2006. Saatavilla www-muodossa http://www.3dkingdoms.com/weekly/weekly.php?a=26 (Luettu 12.8.2010)

Ranta-Eskola, S. 2003. BSP Trees: Theory and Implementation. Saatavilla www-muodossa http://www.devmaster.net/articles/bsp-trees/ (Luettu 20.9.2010).

Samet, H. 1984. The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys 16(2). 187-260.

Shimer, S. 1997. Binary Space Partition Trees. Saatavilla www-muodossa http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html (Luettu 28.9.2010).

Teller, S. 1992. Visibility Computations in Densely Occluded Polyhedral Environments. Saatavilla pdf-muodossa http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-92-708.pdf (Luettu 18.8.2010).

KUVAT

GPWiki. http://gpwiki.org/index.php/Files:Sphere.gif

Povray. http://www.povray.org/documentation/images/reference/sphgeom.png

Slava, V. Bounding Volume. http://3dengine.org/Bounding_volume

WhiteTimberwolf. 2010.
http://upload.wikimedia.org/wikipedia/commons/2/20/Octree2.svg