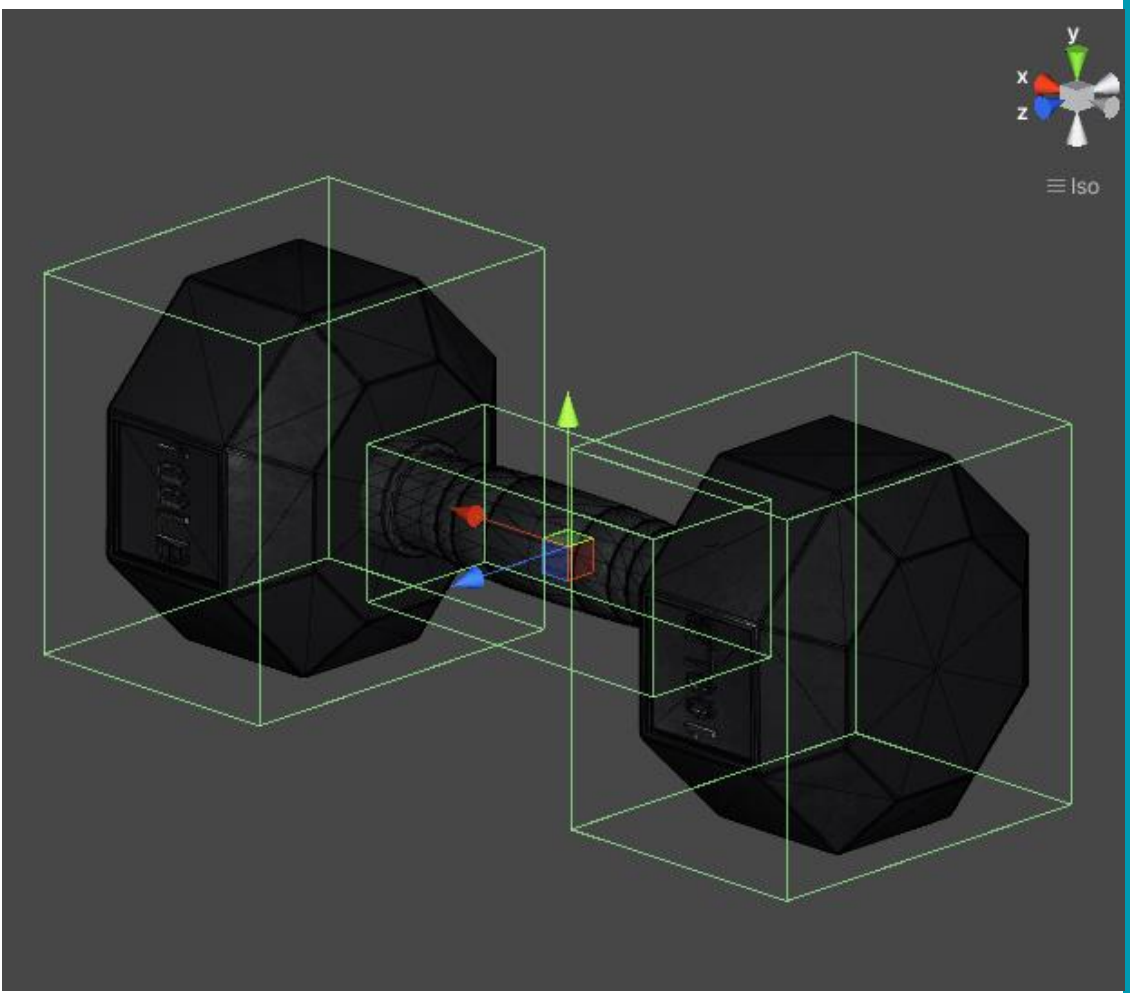
Jere Tuohino

OBB-puun generoiminen Unity Enginessä



Tietojenkäsittely

Tradenomi

Syksy 2021

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Tuohino Jere

Työn nimi: OBB-puun generoiminen Unity Enginessä

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: Rajausalue, OBB, binäärihakupuu, Unity

Tässä opinnäytetyössä kehitetään Unity-pelimoottoriin työkalu, jonka tarkoituksena on luoda OBB-puun teorian mukainen törmäytin annetulle mallille. OBB-puulla pyritään saavuttamaan tarkempaa mallin mukaista törmäytintä, kuin mitä esimerkiksi konveksipeitteellä saadaan konkaavin mallin kanssa.

OBB-puu generoituu jakamalla alkuperäistä OBB-rajausaluetta ja siitä tulleita jakoja. Tätä jatketaan, kunnes yhdessäkään jaossa mallin syvyys alueen sisällä ei muutu annetun raja-arvon ylitse.

Opinnäytetyössä OBB-puun generoimisessa käytetään pohjana Chuhua Xian, Hongwei Lin ja Shuming Gaon kehittämää tutkimusta nimeltä *Automatic cage generation by improved OBBs for mesh deformation*. Tässä tutkimuksessa OBB-puun teko toteutetaan OBB-puun perinteisellä teorialla, poiketen OBB:n jakamisvaiheessa. Perinteisen jakosäännön sijaan tutkimuksessa toteutetaan monimutkaisempi algoritmi, jolla saavutetaan puhtaampi lopputulos.

Opinnäytetyön alussa käydään lävitse työkaluun liittyvät yleiset termit ja käytänteet, kuten rajausalueet, törmäyttimet ja vokselit. Myöhemmin opinnäytetyössä käydään OBB-puun luomisen prosessi askel askeleelta lävitse, avataan sen teoria tarkemmin ja päätetään kertomalla lopputulos, mihin opinnäytetyötä tehdessä päästiin, mitä ongelmakohtia tuli vastaan ja mikä on työkalun tulevaisuus.

Lopputuloksena opinnäytetyöstä koostuu työkalu, jolla pystyy jakamaan testimallina esitetyn 3D-mallin käsipainosta loogisesti kolmeen laatikkoon.

Abstract

Author: Tuohino Jere

Title of Publication: Generating an OBB-tree in Unity Engine

Degree Title: Bachelor of Business Administration, Business Information Technology

Keywords: Bounding box, OBB, Binary search tree, Unity, Voxel

This thesis develops a tool for game development within Unity Engine. The purpose for the developed tool is to generate a collider from an OBB-tree. The goal for using an OBB-tree for the generation of a collider is to achieve a more accurate one than what a mesh collider would offer.

An OBB-tree is generated by dividing the original mesh's OBB and its divisions. This dividing process is continued till no mesh's depth within the divisions changes over a given threshold.

For the generation of the OBB-tree, this thesis follows a study created by Chuhua Xian, Hongwei Lin and Shuming Gao called *Automatic cage generation by improved OBBs for mesh deformation*. The study follows the traditional method of generating an OBB-tree, but differs in the splitting part, where they introduce a new way of splitting the OBBs, which produces more clean results than the traditional way.

The beginning of the thesis focuses on defining the general terms and methods used in the tool, such as bounding boxes, colliders, and voxels. After the theory section the focus shifts toward the development of the tool, going through the steps one by one, and explains the theory a bit more in detail. The last section of the thesis goes through the result, where it can be utilized at that point and what is the future of the tool.

The result from the thesis is a tool that can divide a given 3D-model of a weight logically into three different bounding boxes.

Sisällysluettelo

Symboliluettelo

AABB	Axis Aligned Bounding Box – Akseliriippuvainen törmäytin
C#-ohjelmointikieli	Microsoftin kehittämä ohjelmointikieli
Kolmio	3D-mallin pinnan yksittäinen kolmio
Komponentti	Unity-enginessä skenen objektiin liitetty skripti
Konveksi peite	Konveksin muotoinen törmäytin
Kvaternio	(englanniksi quaternion) Neljästä luvusta koostuva kompleksiarvo
MCSA-funktio	Xian, Lin ja Gaon algoritmissä hyödynnetty funktio nimeltä *Minimum Cross Section Area* -funktioksi
Mesh-data	3D-mallidata, joka sisältää mm. kolmiot, verteksit ja sivut
OBB	Oriented Bounding Box – Oman orientaation omaava törmäytin
OBB-puu	Binääripuuta hyödyntävä törmäytin, joka koostuu useasta OBB:sta
Objekti	Unity Enginessä oleva GameObject
Ohjelmointikirjasto	Valmis kirjasto, joka sisältää useita ohjelmoinnissa hyödynnettäviä funktioita, arvoja ja luokkia
Rajausalue	Pienin mahdollinen alue johon 3D-malli mahtuu
Raycast	Unity Enginessä hyödynnettävä säde, jolla voidaan katsoa törmätyt asiat pisteiden A ja B välillä
Rotaatiokvaternio	Kompleksiluku, joka määrittää rotaation kvaternionista. Poistaa ongelman, joka esiintyy perinteisessä yaw, pitch roll -rotaatiossa
Törmäytin	Yksinkertainen mallin rajausalue, jota hyödynnetään fysiikkalaskujen nopeuttamisessa
Verteksi	3D-mallin kulmapiste
Vokseli	Kolmiulotteinen pikseli
Vokselikuori	Ontto, mallin reunaa seuraava vokseliryhmä
Vokselirivi	Jonkin akselin mukainen rivi vokseleita
Vokseliryhmä	Joukko vokseleita

1 Johdanto

Kahden fyysisen objektin välinen kollision käsittely on ollut ongelma, jonka ratkaisuksi on vuosikymmenien aikana toteutettu monia eri menetelmiä niin videopeleissä että sovelluksissa. Ajan myötä näistä algoritmeista on standardisoitunut useita niiden nopeuden, tarkkuuden ja/tai yksinkertaisuuden takia. Kaksi erittäin yleistä algoritmia kollisolaskuissa pelimoottoreiden sisällä ovat *Axis Aligned Bounding Box* (AABB) ja Konveksi peite. Näistä kahdesta AABB on yksinkertaisin ja täten myös nopein laskea, kun taas konveksi peite antaa tarkemman 3D-meshin mukaisen laskun, mutta on huomattavasti raskaampi tietokoneelle laskea.

Tämä opinnäytetyö keskittyy *OBB-Puu* nimisen törmäytinpuun generoinnin toteuttamiseen, joka on teoriassa nopeudellaan AABB:n ja konveksin peitteen välillä, mutta optimaalisessa toteutuksessa tarjoaa saman tarkkuuden kuin konveksi peite ja joissain tilanteissa jopa tarkemmankin.

Opinnäytetyön tavoitteena on luoda työkalu, joka automatisoi mallin tarkan törmäyttimen generoimisen. Tämä prosessi on työläs tehdä käsin ja täten sen automatisoinnin hyöty on huomattava, etenkin esimerkiksi hyllyjen törmäyttimien yhteydessä, jossa laatikon muotoinen törmäytin ottaa huomioimatta hyllyn sisäisen muodon täysin.

Työkalu toteutetaan oululaiselle yritykselle Peilivision Oy, joka keskittyy neurologiseen kuntoutukseen virtuaalista todellisuutta hyödyntäen. Tilanteessa, jossa jotain objektia pyritään teleportaamaan toiseen pisteeseen, pelin täytyy tietää, mahtuuko objekti kyseiseen tilaan, jolloin OBB-puuta hyödynnetään.

2 Törmäyttimet

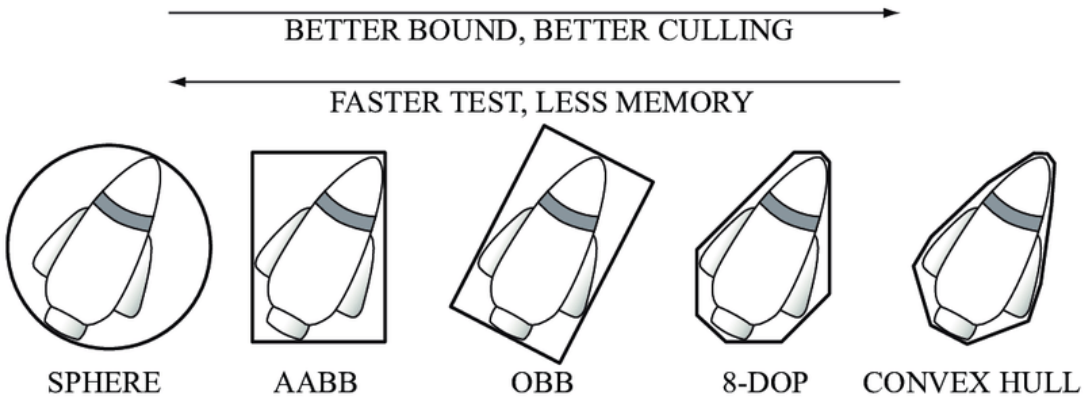
Törmäytin, englanniksi *collider*, on esineen alue, jota hyödyntämällä eri sovelluksissa voidaan määrittää esineiden välisen interaktion ja käyttäjän interaktion muiden esineiden kanssa. Tällainen interaktio voi olla esimerkiksi tilanne, jossa katsotaan, onko pelaajan kursori UI-nappulan päällä.

Jotta tiedettäisiin, osuuko säde johonkin esineeseen, meidän pitäisi käydä lävitse jokainen mallin kolmio tai sivu ja laskea, osuuko säde sen sisälle. Riippuen mallin kompleksisuudesta – etenkin jos malli on kolmiulotteinen – tämä olisi erittäin raskas laskuprosessi tehdä jokainen fysiikkamoottoripäivityksen aikana.

Jotta sovellukset toimisivat sulavasti ja ilman moottorin hidastumista vaadittujen raskaiden törmäyttimiin liittyvien laskujen aikana, on alan normiksi otettu käyttää fysiikkalaskuissa erilaisia primitiivisiä malleja, kuten kuutiota ja palloja [2], tai tarkemman muodon tarvittaessa konveksin muotoista monitahokasta, joka muodostaa esineen rajausalueen.

2.1 Rajausalue

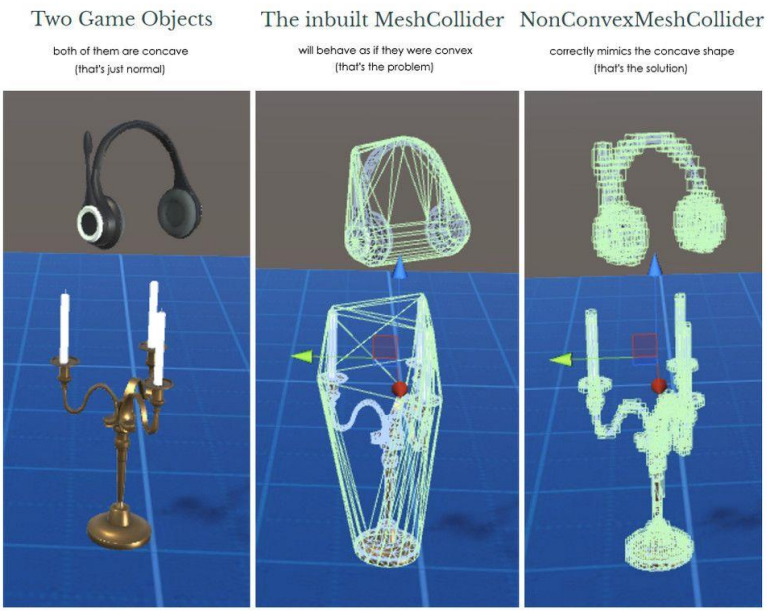
Esineen rajausalue, englanniksi *Bounding Volume*, on pienin mahdollinen laatikko tai pallo, jolle kyseinen esine voi mahtua [3.] Kyseisen rajausalueen määrittämiseen on toteutettu monia eri algoritmeja, joista valitaan tilanteeseen sopiva riippuen tarvittavasta tarkkuudesta ja tehokkuudesta (Kuva 1).



Kuva 1. Eri rajausaluealgoritmit tehokkuus- ja tarkkuusjärjestyksessä. [4]

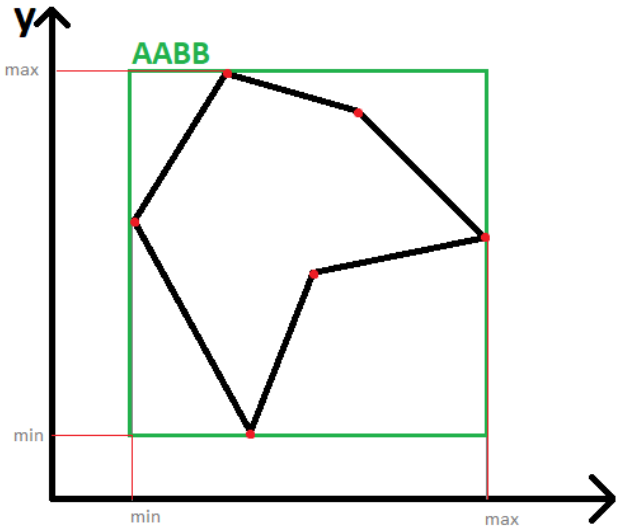
2.2 Konveksi peite

Kuvassa 1 listatuista algoritmeista tarkin on *Convex Hull*, joka suomeksi käännettynä tarkoittaa konveksia peitettä. Yksi tapa kuvailla, miten konveksi peite lasketaan, on kuvitella peitto, joka kääritään tiukasti jonkin esineen ympärille. Tämä yksinkertaistaa kollisiolaskuja kyseiselle objektille paljon, ja mahdollistaa matemaattisten lausekkeiden kuten *Seperating Axis Theorem*:in käyttämisen kahden konveksin muodon interaktion laskemisessa [5]. Tämän algoritmin huono puoli kuitenkin piilee siinä, että kuperista muodoista täytyy muodostaa konveksi versio, joka vaatii syvennyksien poistamista. Tällaisissa tilanteissa konveksi peite peittää mallien reiät tai muut sisäiset muodot. (Kuva 2)



Kuva 2. Konveksin törmäyttimen käyttö kuperassa muodossa ja sen vertaus [6]

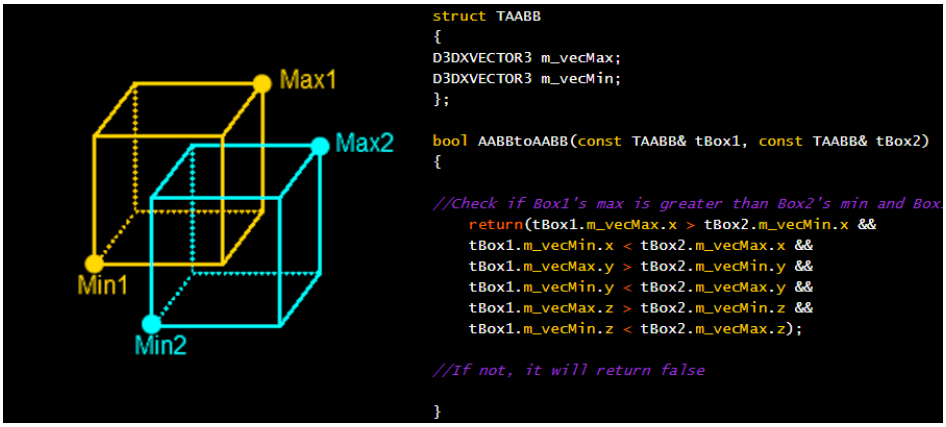
2.3 AABB ja OBB



Kuva 3. AABB piirrettynä kaksiulotteisena

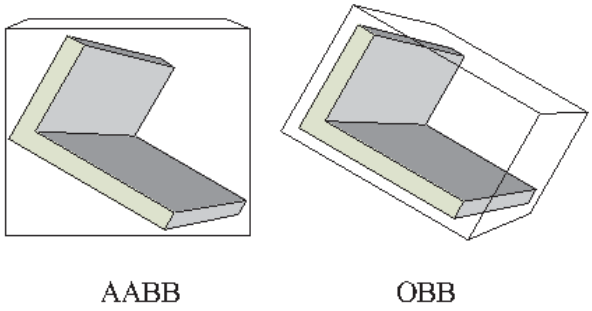
Yksinkertaisin rajausalue heti pallon muodon jälkeen, on *AABB*. Termi AABB muodostuu sanoista *Axis-Aligned Bounding Box*, joka suomeksi käännettynä tarkoittaa ”akselin mukaista rajausaluetta”. Esineen AABB lasketaan käymällä kaikki 3D-mallin pisteet lävitse ja tallennetaan jokaisen akselin mukaan pisteiden pienin ja suurin sijainti kyseisellä akselilla.

AABB on erittäin nopea, sillä kahden AABB:n omaavan objektin välisen interaktion laskemiseen tarvitaan yksi laskutoimenpide, että onko esineen A_{max} suurempi kuin esineen B_{min}, ja että esineen A_{min} on pienempi kuin esineen B_{max} [7.]



Kuva 4. AABB päällekkäisyyden laskeminen [7]

OBB*, eli Oriented Bounding Box,* on rajausalue, joka muistuttaa vahvasti AABB:ta (Kuva 4), mutta poikkeaa sen riippumattomuudesta akseleihin eli se voi orientoitua rotaatioon, millä saavutetaan pienin mahdollinen laatikko pisteiden ympärille. OBB:n kollision laskemiseksi ei voida käyttää samaa vertauslaskua kuin kahden AABB:n välillä, vaan sen sijaan joudutaan käyttämään raskaampaa teoriaa kuten *Seperating Axis Theorem* [5].

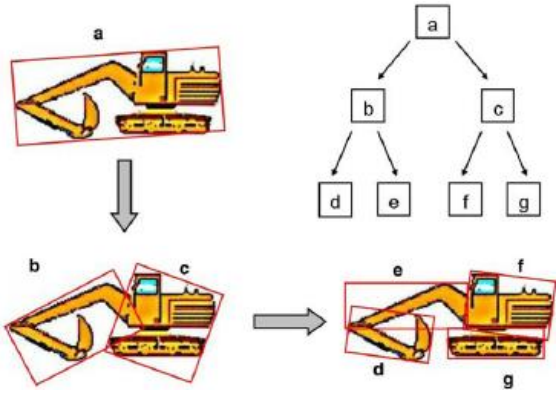


Kuva 5. AABB verrattuna OBB:en [8]

2.4 OBB-puu

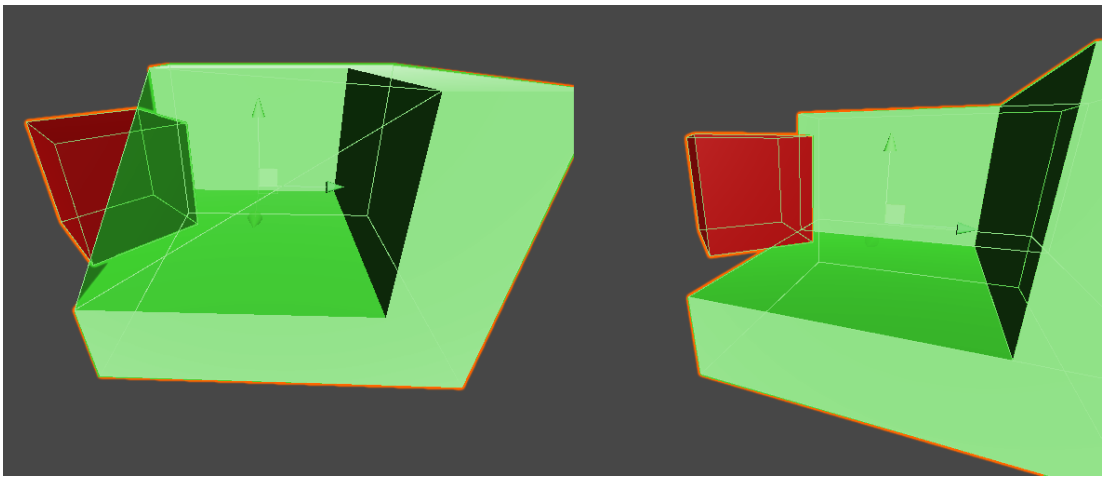
OBB-puun teorian ydin perustuu ohjelmoinnissa käytettyyn algoritmiin nimeltä binäärihakupuu, tai binääripuu lyhyesti. Binääripuu-teoriassa puun aloituskohta – eli juurisolmu – voidaan jakaa enimmäkseen kahteen lapsisolmuun. [9]

OBB-puussa algoritmin juurena – eli aloituspisteenä – käytetään mallin mukaan laskettua alkuperäistä OBB-rajausaluetta, jota jaetaan sopivasta kohdasta kahteen osaan aina jonkin kriteerin täytyttyä. Tällaisia kriteereitä voi olla esimerkiksi, onko malli OBB:n sisällä tarpeeksi *tasainen* vai ei.



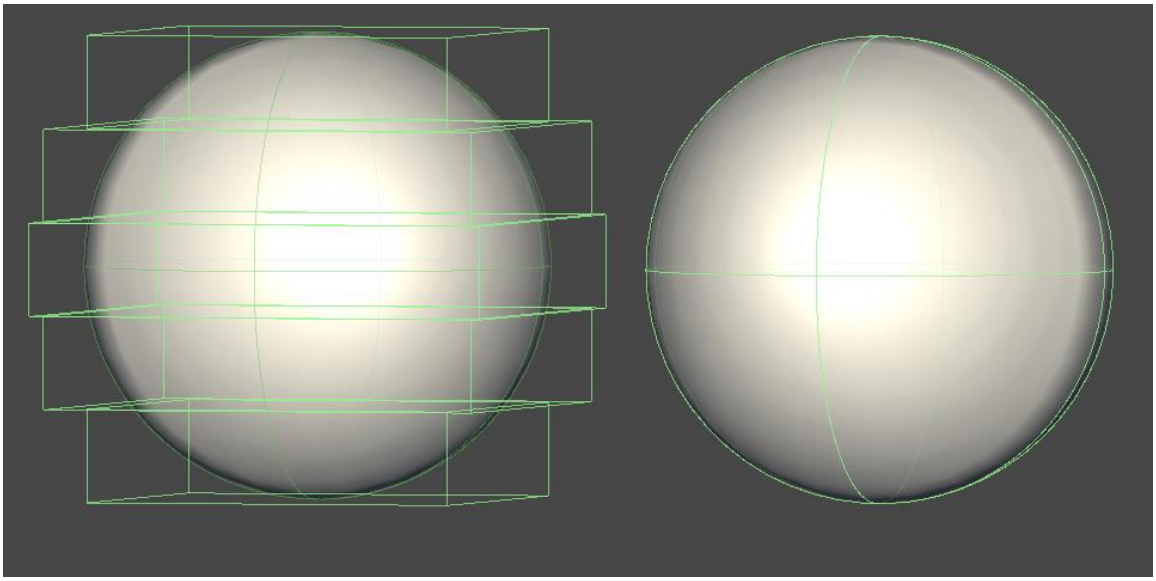
Kuva 6. Esimerkki OBB-puun hyödyntämisestä [10]

Optimaalisessa tilanteessa OBB-puu ottaa huomioon myös kuperat muodot, toisin kuin kaikki kuvassa 1 listatut algoritmit. Tämä tarkkuus voi olla tärkeää esimerkiksi tilanteissa, joissa jokin esine asetetaan toisen kuparin muotoisen esineen viereen ja lasketaan, törmäävätkö ne keskenään. Kuva 6 esittää kyseisestä skenaariosta esimerkkitilanteen. Kuvassa vaaleanvihreät viivat esittävät törmäyttimen alueen ja törmäyttimen sisäinen alue on kuvassa maalattu vihreäksi, jotta alueen visualisointi helpottuu.



Kuva 7. Esimerkki konveksitilanteesta konveksilla peitteellä (vasen) ja OBB-puulla (oikea)

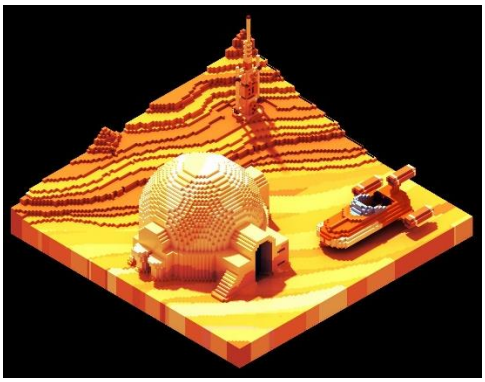
OBB-puu ei kuitenkaan tule olemaan täydellinen ratkaisu kaikkiin 3D-mallien törmäyttimien laskuun. Huonoja tilanteita, joissa OBB-puuta voitaisiin käyttää, on esimerkiksi pyöreät muodot, joiden ympäröiminen laatikoilla ei tule olemaan sen tarkempaa kuin konveksiin peitteen tai pallon muotoisten törmäyttimen käyttäminen.



Kuva 8. Pyöreän muodon rajaaminen OBB-puulla (vasen) verrattuna palloon (oikea)

3 Vokseli

Vokselit yksinkertaisuudessaan ovat pikseleitä, joissa on mukana myös syvyysulottuvuus. 2D-kuvassa pikselin voi käsittää yksikkönä, joka määrittää pienimmän mahdollisimman alueen, kun 2D-avaruus jaetaan erillisiin, samankokoisiin alueisiin. Vokseli on käsitteenä samanlainen, mutta sisältää myös kolmannen ulottuvuuden: syvyyden. [12]



Kuva 9. Antoine Lendrevie:n toteuttama vokselitaideteos [11]

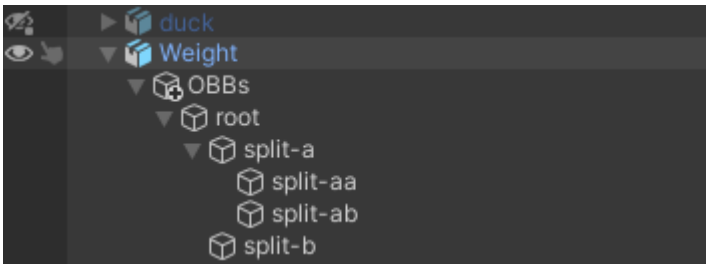
Vokseleita on käytetty esimerkiksi 3D-taiteessa ja esimerkiksi pehmeän kappaleen dynamiikassa (*englanniksi soft-body dynamics*). Tässä opinnäytetyössä toteutetussa työkalussa vokseleita hyödynnetään 3D-mallin rasteroinnissa mallin käyttämän alueen laskemisen ja käsittelyn nopeuttamiseen ja yksinkertaistamiseen.

4 Implementaatio-suunnitelma

Työkalun teoria implementaatiossa perustuu Chuhua Xian, Hongwei Lin ja Shuming Gaon kirjoittamaan tutkimukseen nimeltä ”*Automatic cage generation by improved OBBs for mesh deformation*” [13]. Kyseinen tutkimus hyödyntää samaa teoriaa kuin alkuperäisessä OBB-puun luomisessa [14], mutta poikkeaa siitä OBB:n jakamiskohdassa (lisää kappaleessa 5.3).

Työkalu toteutetaan Unity-engineen C#-ohjelmointikielellä. Unity Engine on pelimoottori, jolla suurin osa viime vuosien aikana julkaistuista peleistä on toteutettu [15]. Syitä tämän pelimoottorin suosiolle ovat sen tarjoamat ilmaiset lisenssit vähän myyville yrityksille, lukuisat yhteisön tekemät oppimateriaalit pelimoottorin käyttämistä varten ja yleensäkin sen helppokäyttöisyys. [16]

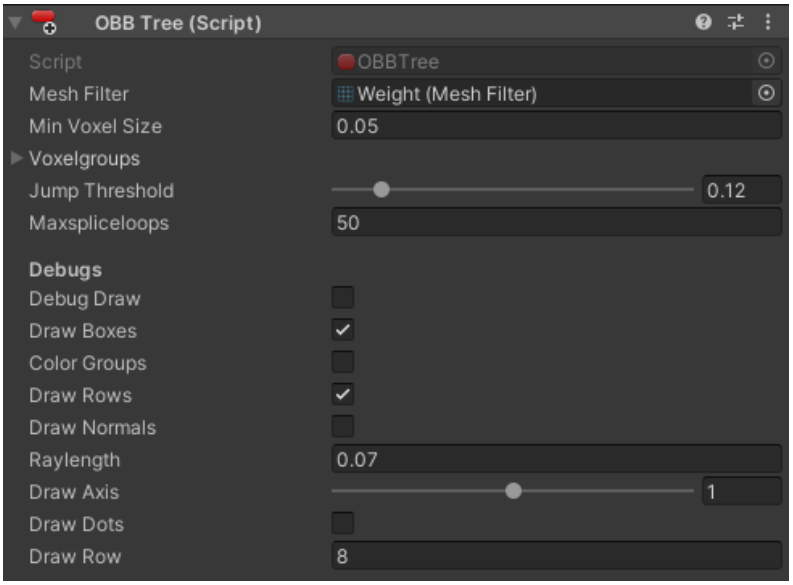
Työkalun tarkoituksena on luoda nappia painamalla monta laatikon muotoista törmäytintä, jotka kokonaisuudessaan rajaavat mallin muodon. Luodut törmäytinlaatikot tulevat *BoxCollider*-komponenteista, jotka sijoitetaan kiinni GameObjekteihin eli peliympäristössä olevin objekteihin.



Kuva 10. Esimerkki Unityn gameobject hierarkiasta

Tämän opinnäytetyön aikana toteutetun työkalun toiminnallisuus rakennetaan täysin komponentin sisällä, jotta koodin testaus ja suuri muokkaaminen olisi mahdollisimman yksinkertaista ja täten nopeaa. Myöhemmin työkalu optimoidaan tuotantovalmiiseen käyttöön, jolloin koodin toiminnallisuus siirretään erilliseen ikkunaan, jossa lopputulosta voidaan esikatsella ennen tallentamista.

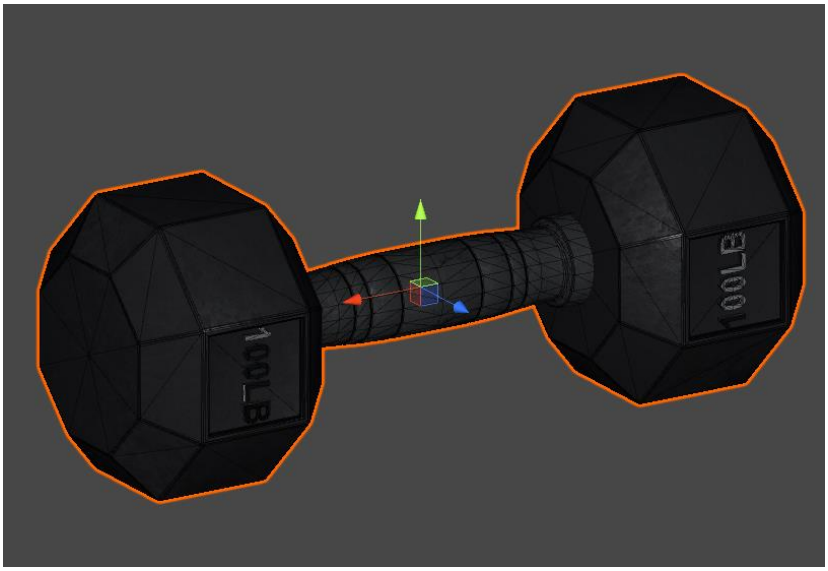
Koska työkalun toiminnallisuus on moniosainen, on tärkeää nähdä, miten työkalu toimii tietyssä osassa. Näin voidaan tarkastella sitä vaihetta, jossa alkuperäistä vokseloitua mallia aletaan vasta jakamaan ryhmiin tai tutkimaan, millä arvoilla kyseinen jako on toteutettu. Työkaluun on myös hyvä kehittää toimintoja, joilla tuotettua tulosta voi tutkia tarkemmin. Esimerkki tällaisesta voisi olla toiminto, joka näyttää vain yhden vokselirivin kerrallaan usean sijaan.



Kuva 11. Työkalun komponentti ja sen arvot

5 Työkalun toteutus

Työkalun testaamista varten tarvitaan malli, joka ei ole konveksin muotoinen. Tällä tavalla työkalun testaamisesta hyödytään eniten. Myös työkalun kehittämistä varten on mallin hyvä olla yksinkertainen vähäisellä muodolla ja kolmiomäärällä, jotta uudelleenprosessointi olisi nopeaa aina pienien ohjelmointimuutoksien jälkeen. Näiden syiden vuoksi päädyin käyttämään 3D-mallia käsipainosta, jonka löysin Sketchfab-nimiseltä sivulta käyttäjän *Blender3D* mallintamana [17].



Kuva 12. Työkalun testaamiseen käytetty 3D-malli [17]

Tämä on sopiva käyttötarkoitukseen, sillä käsipainon varren sivuilla olevat painot ovat huomattavasti suuremmat varteen verrattuna ja ne muodostavat mallin keskelle varren ympärille tyhjän tilan. Kyseinen tyhjä tila aiheuttaa tilanteen, jossa yhden laatikkotörmäyttimen sisällä tyhjä tila laskettaisiin osaksi mallia. Lisäksi käsipainon jako OBB-puuksi on yksinkertainen toteuttaa käsin, ja sitä voi sitten helposti verrata algoritmin toteuttamaan lopputulokseen nähdäkseen, kuinka hyvin algoritmi optimaalisessa tilanteessa toimii.

5.1 Mallin OBB:n laskeminen

Koska OBB:n laskeminen on kompleksi laskutoimenpide toteuttaa ja tämä tutkimus keskittyy OBB-puun toteuttamiseen eikä yksittäisen OBB:n, hyödynnetään kehitysvaiheessa valmista ohjelmointikirjastoa nimeltä *MathGeoLib* [18].

MathGeoLib on C++-ohjelmointikirjasto, joka tarjoaa lineaarialgebraan ja geometrian manipulointiin liittyviä funktioita ohjelmoijan käytettäväksi. Tässä tutkimuksessa hyödynnetään kirjaston pienimmän OBB:n laskemiseen tarjottua funktiota, joka on toteutettu Jukka Jylängin tutkimuksessa [19].

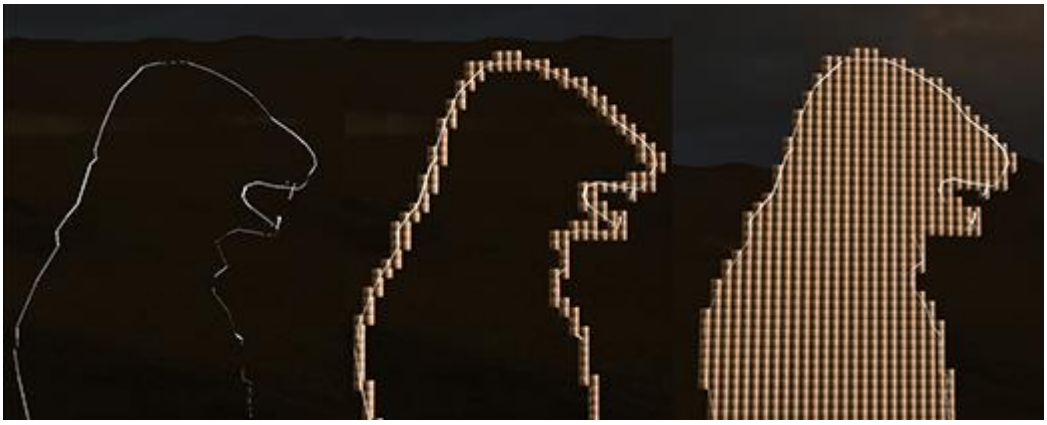
Koska Unity Enginessä ohjelmointi toteutetaan C#-kielellä, on kyseiselle MathGeoLabistä löytyneelle OBB:n laskemisen ja sen käsittelyn funktioille toteutettu kääre. Sen avulla voi kutsua kirjaston funktioita C#-kielellä ja kääntää tulokset Unity pelimoottorille sopiviksi [20].

Jotta saadaan pienin OBB:n mallin, syötetään MathGeoLab:in funktioon mallin kaikki verteksit, joka tuo funktiosta OBB:n kolme kantavektorin, keskikohdan ja ulottuvuuden sisältävän luokan. Näiden kolmen kantavektorin avulla voidaan määrittää OBB:n rotaatio kvaternio (*englanniksi quaternion*), joka annetaan OBB:n omaavalle objektille. Luokan keskikohdalla ja ulottuvuudella voidaan luoda OBB:n mukainen Box Collider -komponentti oikeilla arvoilla objektille.

5.2 Mallin vokselointi

Työkalun mallin vokselointivaiheessa seurattiin David Rosenin kirjoittamaa blogipostausta Wolfire Games -sivulta [21]. Vokselointi aloitetaan jakamalla objektin OBB tietyn kokoisiin kuutioihin. *Automatic cage generation by improved OBBs for mesh deformation* -tutkimus [13] määrittää vokselin koon kahden lähimmän verteksin etäisyyden mukaan, mutta sen sijaan tässä työkalussa vokselin koon määrittävän muuttuja annetaan kehittäjän säädettäväksi julki, jotta kehitysvaiheessa lopputulosten testaaminen olisi mahdollisimman nopeaa.

Työkalun jaettua 3D-mallin alkuperäisen OBB:n vokseleihin, säästetään niistä ne, jotka sisältävät osan mallia. Tästä muodostuu niin sanottu vokselikuori. Vokselikuoren laskeminen on melko yksinkertainen prosessi: kaikki mallin kolmiot käydään listassa lävitse, ja kaikki kolmion AABB:n sisällä olevat vokselit testataan, sisältävätkö ne osan kolmiosta. Jos vokseli sisältää osan kolmiosta, se lisätään osaksi vokselikuorta.



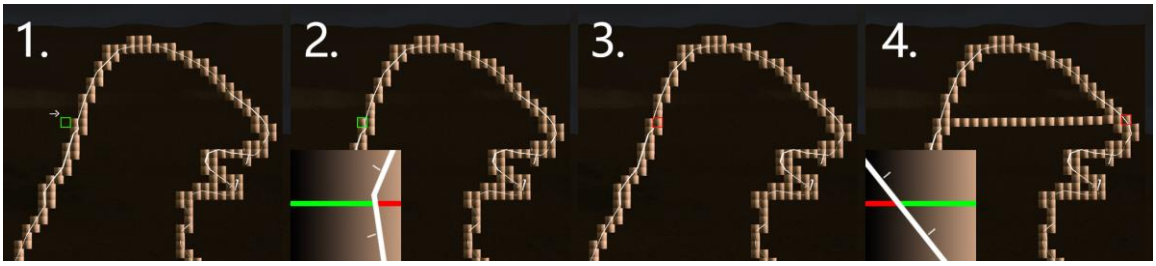
Kuva 13. Mallin reunojen (vasen) mukaan luotu vokselikuori (keskellä) ja täytettynä (oikealla) [21]

Työkalun laskettua mallin vokselikuoren, se pitää vielä täydentää sisäpuolelta, jotta se ei ole ontto. Tämän toteuttamiseen David Rosen hyödyntää niin sanottua kynämenetelmää. Siinä vokselikuorta käydään lävitse rivi kerrallaan, jokainen vokseli käydään lävitse rivin kulkevaan suuntaan ja testataan, sisältääkö se mallin kolmioita. Kolmion löydyttyä katsotaan vokselin sisällä olevien kolmioiden normaalit, ja lasketaan normaalien avulla, onko testatun rivin kulkemissuunta astumassa mallin sisälle vai ulos.

David Rosenin kirjoittamassa mallin vokselointiprosessissa hän katsoo, onko hän tietyssä poistumassa tai astumassa mallin sisälle hyödyntämällä mallin lävistävää Raycastia. Tässä kohti työkalua poikkesin hänen suunnitelmastaan, ja Raycastin sijaan hyödynnän vokselikuoren laskemisvaiheessa tallennetun jokaisen vokselin sisällä olevien kolmioiden normaaleita.

Tietääksemme, onko suunta menossa mallin sisään tai ulos mallista, laskemme vokselin sisällä olevien kolmioiden normaalien ja kulkemissuunnan pistetulon ja vertaamme tulosta lukuihin 1 ja -1. Jos pistetulon arvo on negatiivinen (-), tiedämme, että kolmioiden suunta on kulkemissuuntaa vastaan, joten olemme astumassa mallin sisälle. Jos arvo taasen onkin positiivinen (+), tiedämme vokselin sisällä olevien kolmioiden normaalien olevan menosuunnan mukaiset, jolloin olemme astumassa mallista ulos.

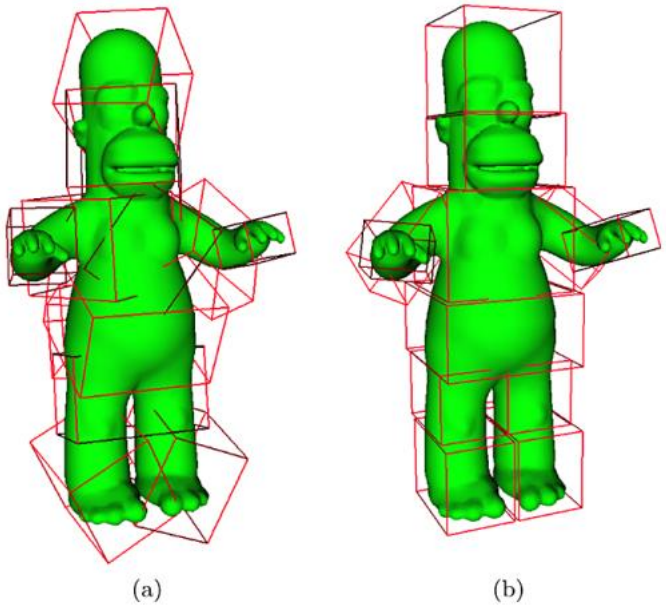
Kynämenetelmässä hyödynnämme kyseistä tietoa siten, että aina kun mallin sisälle astutaan, kynä lasketaan alas ja rivin läpikuljettuihin ruutuihin luodaan uusia vokseleita. Ulos astuessa kynä nostetaan ylös ja täyttö lopetetaan.



Kuva 14. Vokselikuoren täyttö, verraten kolmioiden normaaleihin onko ”kynä” menossa mallin sisään (2.) vai ulos mallista (4.) [21]

5.3 Vokseliryhmän jako kahteen

Alkuperäisessä OBB-puun toteutuksessa OBB jaettiin sen pisimmän sivun mukaisesti keskeltä kahtia, jos malli ja jaettava OBB täyttää tietyt kriteerit [14]. Xian, Lin ja Gaon tutkimuksessa toteutettu algoritmi kehittää tätä jakoalgoritmia eteenpäin tarkemmaksi ja tehokkaammaksi [13].



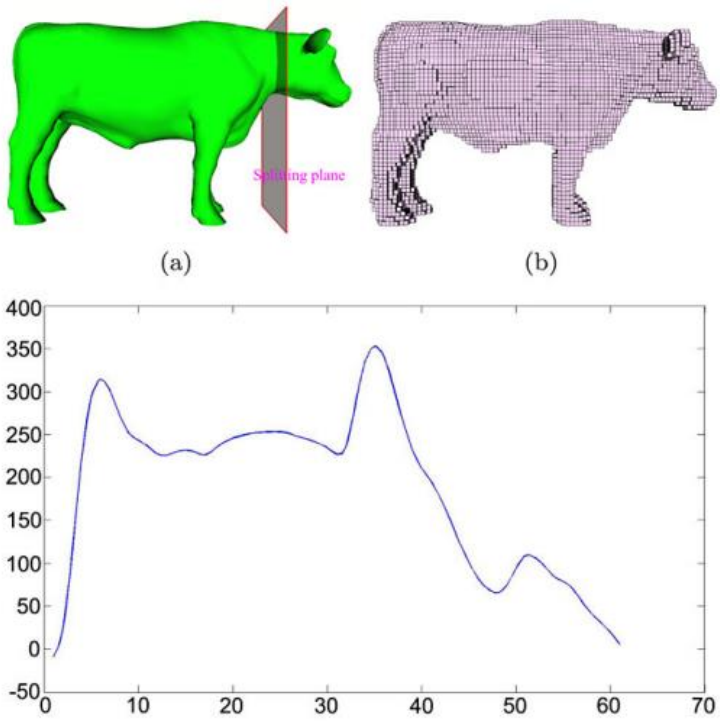
Kuva 15. Perinteinen jakoalgoritmi (a) verrattuna kehitettyyn algoritmiin (b) [13]

Xian, Lin ja Gaon kehittämässä algoritmissa käytetään funktiota, jota he kutsuvat nimellä *Minimum Cross Section Area* -funktioksi. Tässä opinnäytetyössä kyseistä funktiota kutsutaan jälkeenpäin MCSA-funktiona. MCSA-funktion tarkoituksena on palauttaa arvo, mikä on mallin pienin syvyys jollain akselilla kohdassa *x*.

Käyttääksemme MCSA-funktiota, malli pitää jakaa vokseleihin ja verrata, miten syvä malli on akselilla kohdalla *x*. Jos akselin testatussa kohdassa malli ei ole yhtenäinen, kuten esimerkiksi kohta

joka sisältäisi ihmismallin kädet ja kehon, otetaan huomioon vain pienin leikkausalue, joka annetussa esimerkissä todennäköisesti olisi yksi käsistä. Muissa tapauksissa funktion tuloksena tulee tämän kohdan syvyys.

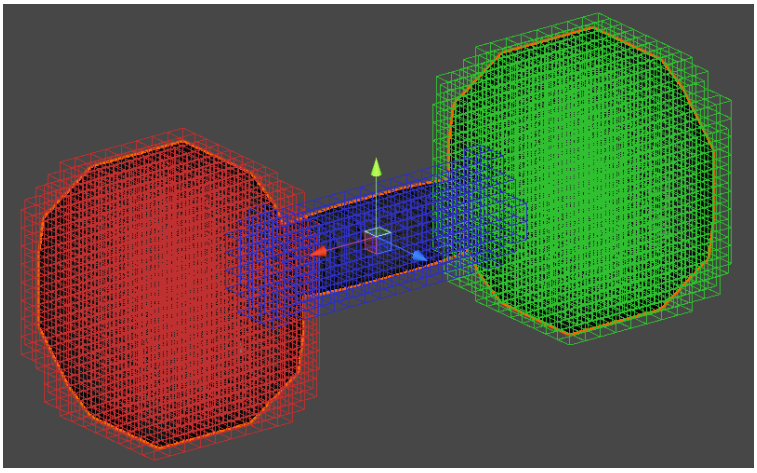
Kun koko malli on käyty läpi yhdellä akselilla MCSA-funktiolla, saaduista tuloksista katsotaan kohta, missä suurin hyppy tapahtuu viereisiin kohtiin verrattuna. Tämä lasketaan yksinkertaisella jakolaskulla, jossa testataan kohta *x*, ja jaetaan se sen naapuririvillä, joka omaa suurimman syvyyden.



Kuva 16. Kehitetty OBB:n leikkaussääntö (a) lehmämalli ja kohta, missä on suurin muutos muodossa. (b) Vokseloitu lehmämalli (c) MCSA-funktion generoima kaavio [13.]

Algoritmin MCSA-funktiossa käydään lävitse ensimmäisenä OBB:n suurin sivu ja katsotaan, toteutuuko kyseisellä sivulla sopivaa hyppykohtaa, joka olisi työkalussa määritetyn rajan yllä. Kun leikkauskohta on löydetty, vokseliryhmä jaetaan kyseisestä kohdasta kahteen ja OBB-puussa lisätään nämä jakaumat jaetun OBB:n alle kahdeksi OBB-lehdeksi.

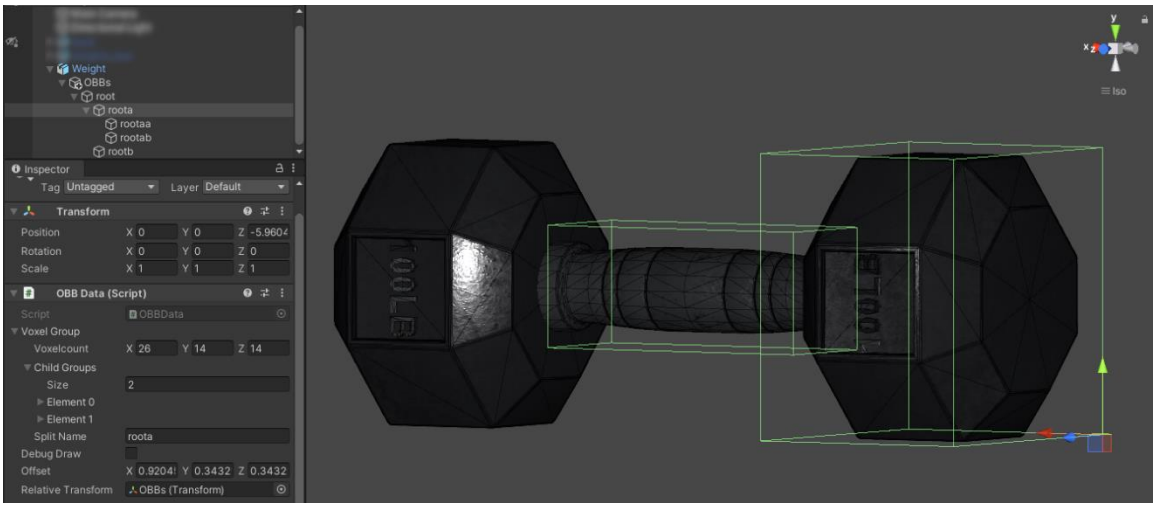
Jos kyseisellä pisimmällä sivulla ei kyseistä hyppyä toteudu, katsotaan seuraavana toiseksi pisin sivu lävitse ja jos se epäonnistuu, katsotaan OBB:n lyhyin sivu. Jos mikään sivuista ei sisällä rajan ylittävää hyppykohtaa, todetaan ettei OBB:ta tarvitse jakaa sen enempää ja OBB-puu on valmis.



Kuva 17. Vokselimalli jaettu vokseliryhmiin. Eri vokseliryhmät värikoodattuna

5.4 Vokseliryhmien OBB:t ja lopputulos

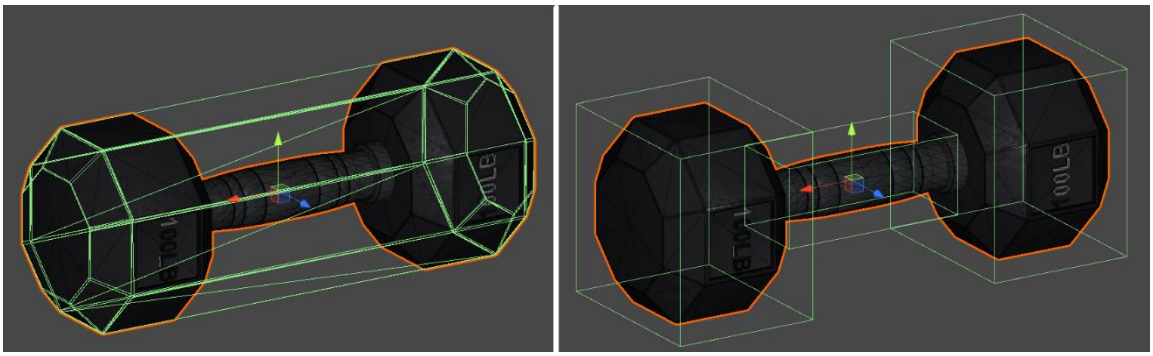
Kun alkuperäinen OBB on käyty lävitse ja todetaan, ettei sen jaoista ja niiden jaoista pystytä jakamaan enempää, luodaan Unityssä jokaiselle lehdelle oma GameObject, joka omaa komponentin, johon on tallennettu sen vokseliryhmä ja siihen liittyvä data. Viimeisille jaoille lasketaan vokseliryhmän mukainen OBB samalla MathGeoLabin funktiolla kuin mitä käytimme alkuperäiselle mallille mallin OBB:n laskemisessa. Nyt mallin verteksien sijasta funktiolle syötetään vokseliryhmän vokseleiden pisteet.



Kuva 18. Generoidun GameObject OBB-puun hierarkia ja OBB Data -komponentti

6 Yhteenveto

Työkalulla toteutettu OBB-puu käsipainosta muodostuu kuin suunniteltu, ja se muodostuu odotetussa järjestyksessä. Kun muodostettua OBB-puuta verrataan esimerkiksi konveksiin peitteeseen, OBB-puu on odotetusti tarkempi mallin keskikohdassa. Tämän perusteella voi todeta käytetyn algoritmin toimivaksi konseptitodisteena.



Kuva 19. Konveksi peite verrattuna työkalun luomaan OBB-puuhun

Kehitetty työkalu ei kuitenkaan ole täydellinen, ja koodi sisältää useita ongelmakohtia, jotka pitää korjata ennen sen hyödyntämistä tuotekehityksessä. Näitä ongelmakohtia ovat muun muassa työkalun hitaus ja lukuisat mysteeriset ongelmat pilkkomisvaiheessa, jotka todennäköisesti johtuvat jostain ohjelmointivirheestä, joiden paikantamiseen ja korjaamiseen opinnäytetyötä tehdessä aika ei riittänyt.

6.1 Työkalun hitaus

Työkalua ei voida hyödyntää pelaamisen aikana sen hitauden takia, vaan sillä kannattaa laskea OBB-puut etukäteen ja tallentaa pelin ympäristöön objektiin suoraan, mutta myös tähänkin tarkoitukseen työkalu on erittäin raskas käyttää kyseisessä muodossa.

Kaaviossa 1 näkyy taulukko, johon on listattu työkalun keston tietyissä prosessin vaiheissa. Ajat on otettu ylös työkalun käsitellessä käsipainoa, joka ei pitäisi olla erittäin raskas käsitellä. Käsipainossa on 3136 kolmiota, ja käytetyssä jaossa jokaisen vokselin kooksi annettiin 0.05 yksikköä, joka jakoi mallin OBB:n 37x14x14 vokselikuutioon, eli yhteensä 7 252 vokseliin.

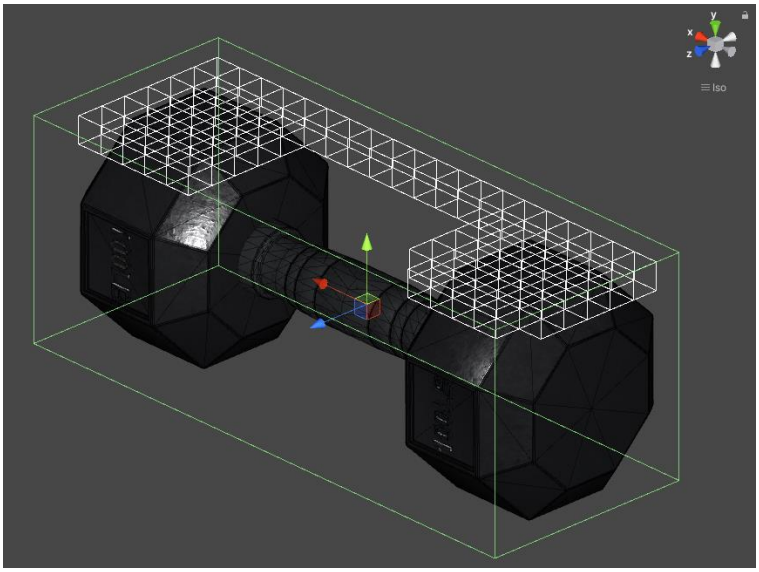
	OBB:n laskeminen	OBB:n vokselointi	Vokselikuoren teko	Vokselikuoren täyttö	Vokselikuoren jako
Kesto (s)	0,002426147	0,018753050	15,021160000	0,501739500	0,034698490
Kesto (%)	0,02 %	0,12 %	96,42 %	3,22 %	0,22 %

Kaavio 1. Työkalun nopeus käsipainon OBB-puun laskemisessa.

Raskain prosessi kaikista vaiheista on ehdottomasti vokselikuoren laskeminen, jonka laskussa kesti yhteensä 15 sekuntia. Tämä todennäköisesti johtuu huonosti optimoiduista funktiosilmukoista, sillä kyseisessä vaiheessa kaikki käsittelemättömät vokselit verrataan kaikkiin mallin kolmioihin, ja tallennetaan ne, mitkä sisältävät jonkin kolmion sisällään. Tämän korjaamiseksi jokaisen kolmion kohdalla olisi todennäköisesti nopeampaa laskea kolmion AABB:n alueen sisällä olevat vokselit ja käsitellä vain ne kaikkien vokseleiden sijasta.

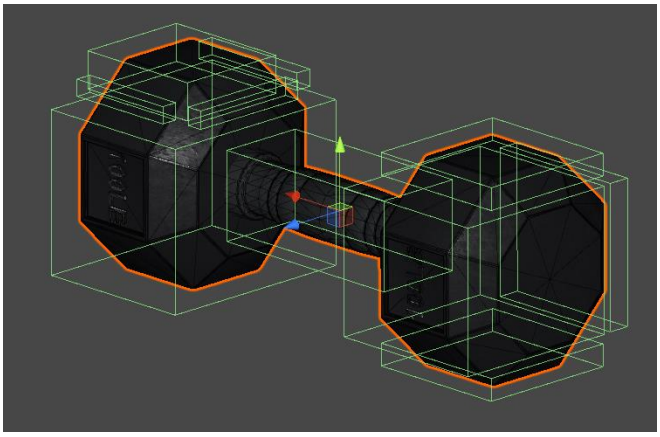
6.2 Ongelma jakamisessa

Työkalun vaiheessa, jossa vokselikuoren välit täytetään, saattaa myös esiintyä ongelma, jolloin vokselikuoren vokselin sisällä olevat kolmioiden normaalien keskiarvo osoittavat mallin sisäpuolelle, jolloin täyttäessä vokselikuorta algoritmi ei tunnista poistuvansa mallista ulos. Tämä ongelma on todennäköisesti korjattavissa siten, että kolmioiden normaalien sijaan hyödynnän Raycast-sädettä, joka katsoo liikkumismäärän pituisella säteellä, meneekö se mallin sisälle tai ulos, kuten viitatussa David Rosenin blogissa ohjattiin.



Kuva 20. Ongelma vokselikuoren täytössä

Riippuen työkalussa määritetystä vokselikoosta ja hyppyrajasta, työkalu saattaa tunnistaa kahden vokselin verran olevan muutoksen vokseliryhmän syvyydessä tarpeeksi suurena jakaakseen vokseliryhmän. Tämä aiheuttaa sen ylimääräisen pirstoutumiseen ja epäoptimiin tulokseen.



Kuva 21. Työkalun yliherkkyys jakoprosessissa

Näiden ongelmakohtienkin kanssa, tässä opinnäytetyössä kehitetyn työkalun lopputulos on omasta mielestäni siltikin tyydyttävä, sillä listatut ongelmat johtuvat todennäköisemmin omasta prototyypin toteutuksesta kuin varsinaisesta teoriasta, johon työkalu perustuu ja täten korjattavissa tuotantokäyttöön varten. Pienen optimoinnin ja ongelmanetsinnän jälkeen työkalusta voi tulla erittäin iso hyöty esimerkiksi kaappien törmäyttimien tekemisessä, joka normaalisti vaatii paljon käsityötä pelimoottorin sisällä.

Lähteet

1. Manocha, Dinesh & Lin, Ming & Brooks, Frederick & Gottschalk, Stefan. (2000). Collision Queries using Oriented Bounding Boxes. Haettu 11.10.2021, osoitteesta: http://gamma.cs.unc.edu/users/gottschalk/main.pdf

2. Baptiste Angles. Geometric modeling with primitives. Networking and Internet Architecture [cs.NI]. Université Paul Sabatier - Toulouse III, 2019. Haettu 12.10.2021, osoitteesta https://tel.archives-ouvertes.fr/tel-02896431/document

3. Introduction to Acceleration Structures: Bounding Volume, Scratchapixel. Haettu 12.10.2021, osoitteesta https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume

4. Mei, Gang. (2014). RealModel-a system for modeling and visualizing sedimentary rocks. Haettu 12.10.2021, osoitteesta https://www.researchgate.net/publication/272093426_RealModel-a_system_for_modeling_and_visualizing_sedimentary_rocks

5. SAT (Seperating Axis Theorem). Haettu 13.10.2021, osoitteesta https://dyn4j.org/2010/01/sat/

6. Non Convex Mesh Collider tool, Unityfreaks. Haettu 13.10.2021, osoitteesta https://unityfreaks.com/asset/3509

7. Miguel Casillas, AABB to AABB. Haettu 13.10.2021, osoitteesta http://www.miguelcasillas.com/?p=30

8. Melissa Angelini, How Collision Detection works (v2020). Haettu 18.10.2021, osoitteesta https://devdept.zendesk.com/hc/en-us/articles/360011559320-How-Collision-Detection-works-v2020

9. Jarmo Siltaneva, Satunnaisten binääripuiden generointi. Haettu 18.10.2021, osoitteesta https://trepo.tuni.fi/bitstream/handle/10024/87877/Siltaneva_Jarmo.pdf

10. Sulaiman, Hamzah & Bade, Abdullah. (2012). Bounding Volume Hierarchies for Collision Detection. 10.5772/35555. Haettu 18.10.2021, osoitteesta https://www.researchgate.net/publication/224829148_Bounding_Volume_Hierarchies_for_Collision_Detection

11. Antoine Lendrevie, ”Did a little Starwars scene”, Twitter. Haettu 19.10.2021, osoitteesta https://twitter.com/sir_carma/status/582235874081050624

12. Matej Jan, Pixels and voxels, the long answer. Haettu 19.10.2021, osoitteesta https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190

13. Xian, Chuhua & Lin, Hongwei & Gao, Shuming. (2012). Automatic cage generation by improved OBBS for mesh deformation. The Visual Computer. 28. 21-33. 10.1007/s00371-011-0595-6. Haettu 20.10.2021, osoitteesta https://www.researchgate.net/publication/220067720_Automatic_cage_generation_by_improved_OBBS_for_mesh_deformation

14. S. Gottschalk, M. C. Lin & D. Manocha, (1996) OBBTree: a hierarchical structure for rapid interference detection. Haettu 25.10.2021, osoitteesta https://dl.acm.org/doi/abs/10.1145/237170.237244

15. Lars Doucet, Anthony Pecorella, Game engines on Steam: The definitive breakdown. Haettu 19.10.2021, osoitteesta https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown

16. Vivek Shah, Reasons Why Unity3D Is So Much Popular In The Gaming Industry, https://medium.com/@vivekshah.P/reasons-why-unity3d-is-so-much-popular-in-the-gaming-industry-705898a2a04

17. Simple Weight, Sketchfab. Haettu 25.10.2021, osoitteesta https://sketchfab.com/3d-models/simple-weight-12cf6e663d214cecaa0885098fea11cd

18. MathGeoLib, Github. Haettu 25.10.2021, osoitteesta https://github.com/juj/MathGeoLib

19. Jukka Jylänki, An Exact Algorithm for Finding Minimum Oriented Bounding Boxes. Haettu 25.10.2021 http://clb.confined.space/minobb/minobb.html

20. MathGeoLib.Exports, Github. Haettu 25.10.2021, osoitteesta https://github.com/aybe/MathGeoLib.Exports

21. David Rosen,Triangle mesh voxelization. Haettu 26.10.2021, osoitteesta http://blog.wolfire.com/2009/11/Triangle-mesh-voxelization