hyväksymispäivä arvosana

arvostelija

Yksinkertaisen monikulmion kolmiointialgoritmit

Teppo Niinimäki

Helsinki 25.4.2008

Tieteellisen kirjoittamisen kurssin tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Teppo Niinimäki

Työn nimi — Arbetets titel — Title

Yksinkertaisen monikulmion kolmiointialgoritmit

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Tutkielma	25.4.2008	21 sivua

Tiivistelmä — Referat — Abstract

Tutustumme erilaisiin yksinkertaisen monikulmion kolmiointialgoritmeihin. Esitämme tavan kolmioida yksinkertainen monikulmio $O(n \log n)$-ajassa. Algoritmissa jaetaan monikulmio aluksi monotonisiin osiin ja kolmioidaan tämän jälkeen saadut osat lineaarisessa ajassa. Lisäksi tutustumme monikulmion puolisuunnikashajotelmaan, ja esittelemme kolmiointiin sitä hyödyntävät aikavaativuuden $O(n \log^* n)$ satunnaisalgoritmin sekä aikavaativuuden $O(n \log \log n)$ deterministisen algoritmin. Lopuksi luomme katsauksen kolmiointiin lineaarisessa ajassa.

ACM Computing Classification System (CCS):
I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling
F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Geometrical problems and computations

Avainsanat — Nyckelord — Keywords

monikulmion kolmiointi, puolisuunnikashajotelma

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Yksinkertainen kolmiointialgoritmi **2**

2.1 Monikulmion jakaminen monotonisiin osiin 3

2.2 Monotonisen monikulmion kolmiointi 6

3 Puolisuunnikashajotelma **9**

4 Nopea satunnaisalgoritmi kolmiointiin **10**

4.1 Yksinkertainen $O(n \log n)$-aikainen algoritmi 11

4.2 $O(n \log^* n)$-aikainen laajennus 13

5 Nopea deterministinen kolmiointi **14**

6 Kolmiointi lineaarisessa ajassa **18**

6.1 Deterministinen algoritmi 18

6.2 Satunnaisalgoritmi 19

7 Yhteenveto **19**

Lähteet **20**

1 Johdanto

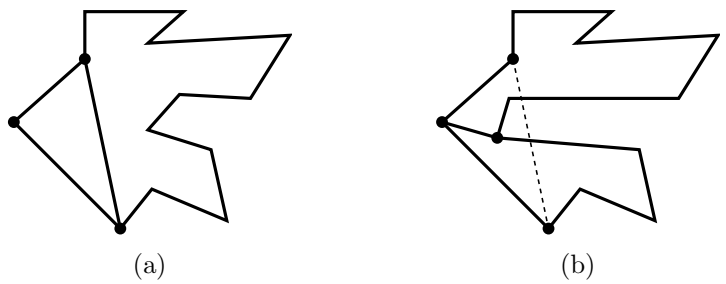
Monikulmion kolmioinnilla tarkoitetaan n-kulmaisen monikulmion sisäosan pilkkomista erillisiin kolmioihin halkaisemalla se kulmien välille vedettävillä lävistäjillä. Lävistäjiä tarvitaan $n-3$ kappaletta, ja tällöin syntyy $n-2$ kolmiota.

Yleisesti käytetty esimerkki monikulmion kolmioinnin käyttökohteista on ns. *taidegallerian vartiointiongelma* [dBvKOS00, Jos94], jossa päämääränä on kattaa valvontakameroilla useasta huoneesta koostuva taidegalleria kokonaan. Ongelmana on valita kameroille sellaiset sijainnit, että niitä tarvitaan mahdollisimman vähän, mutta kuitenkin niin, että jokaiseen gallerian kolkkaan on näköyhteys vähintään yhdestä kamerasta. Galleria pohjapiirros muodostaa monikulmion. Jos monikulmion kolmiointi on tiedossa, voidaan lineaarisessa ajassa laskea $\lfloor n/3 \rfloor$ kameran ratkaisu [dBvKOS00]. Tämä on pahimman tapauksen optimi.

Muita kolmioinnin mahdollisia käyttokohteita ovat esimerkiksi monikulmion sisällä kulkevan lyhimmän reitin laskeminen eri pisteiden välillä sekä pisteiden näkyvyysongelmat monikulmion sisällä [GHL+87]. Tietokonegrafiikassa myös erilaisten pintojen nopea renderöinti tapahtuu usein jakamalla pinta kolmioiksi, joiden piirtäminen voidaan laitteiston avulla hoitaa tehokkaasti [Hel01].

Monikulmiota sanotaan *yksinkertaiseksi*, jos sitä rajaa yksi itseään leikkaamaton reunapolku. Näin ollen se koostuu yhtenäisestä alueesta eikä sisällä reikiä. Yksinkertainen monikulmio $P = \{p_1, p_2, \ldots, p_n\}$ määritellään luettelemalla sen reunapolun kulmapisteet $p_1, p_2, \ldots, p_n$ polun mukaisessa järjestyksessä vasta- tai myötäpäivään. Tässä keskitymme yksinkertaisen monikulmion kolmiointiin erilaisilla algoritmeilla.

Luvussa 2 esitämme yksinkertaisen $O(n \log n)$-aikaisen algoritmin kolmioinnin muodostamiseksi. Algoritmin koostuu monikulmion jakamisesta monononisiin osiin ja näiden osien lineaariaikaisesta kolmioinnista. Luvussa 3 esittelemme monikulmion puolisuunnikashajotelman ja monotonisiin osiin jakamisen sen avulla. Luvuissa 4 ja 5 käymme läpi $O(n \log^* n)$-ajan satunnaismenetelmän ja $O(n \log \log n)$-ajan deterministisen menetelmän kolmiointiin. Lopuksi luvussa 6 luomme katsauksen kolmiointiin lineaarisessa ajassa.



Kuva 1: (a) Kolme peräkkäistä reunapistettä muodostavat korvan. (b) Kolme peräkkäistä reunapistettä eivät muodosta korvaa, jolloin voidaan lisätä lävistäjä korvaehdokkaan keskimmäisen kulmapisteen ja sitä lähimmän korvan sisällä olevan pisteen välille. (Pohjautuu de Bergin ja kumppaneiden [dBvKOS00] kuviin.)

2 Yksinkertainen kolmiointialgoritmi

Suoraviivainen lähestymistapa suorittaa monikulmion kolmiointi on leikata kolmioita irti *korva* kerrallaan [Hel01]. Tässä korvalla tarkoitetaan sellaista kolmen peräkkäisen kulmapisteen muodostamaa kolmiota, joka sisäosa on kokonaan monikulmion sisäpuolella (kuva 1a). Korvia täytyy poistaa $O(n)$ kappaletta, kullakin poistolla on $O(n)$ korvaehdokasta testattavana ja yhden ehdokkaan testaamiseen kuluu $O(n)$-aika. Näin ollen tämä triviaali algoritmi johtaa pahimman tapauksen aikavaativuuteen $O(n^3)$.

Edellisen algoritmin aikavaativuus voidaan pudottaa luokkaan $O(n^2)$ muuntamalla sitä hieman [dBvKOS00]. Muunnoksessa riittää testata vain yhtä korvaehdokasta. Ehdokkaaksi valitaan vasemmanpuoleisimman kulmapisteen ja sen naapureiden muodostama kolmio. Jos ehdokas on korva, voidaan se leikata irti ja jatkaa jäljellä olevalla osalla (kuva 1a). Jos ehdokas ei ole korva, pystytään lineaarisessa ajassa löytämään lävistäjä, joka jakaa monikulmion kahteen osaan (kuva 1b). Tämän jälkeen algoritmia kutsutaan rekursiivisesti kummallekin osalla.

Neliöllisen aikavaativuuden alapuolelle pääsivät ensimmäisinä Garey ja kumppanit [GJPT78]. Esittelemme tämän ajan $O(n \log n)$ vaativan kolmiointialgoritmin yksinkertaisille monikulmioille. Algoritmissa monikulmio jaetaan ensin helpommin käsiteltäviksi alimonikulmioiksi, joita sanotaan monotonisiksi. Tämän jälkeen kolmioidaan syntyneet monotoniset monikulmiot.

2.1 Monikulmion jakaminen monotonisiin osiin

Yksinkertainen monikulmio P on *monotoninen suoran ℓ suhteen*, jos minkä tahansa ℓ:n kanssa kohtisuorassa olevan suoran ja monikulmion P leikkaus on yhtenäinen. Leikkaus on siis jana, piste tai tyhjä joukko. Monikulmio on *y-monotoninen*, jos se on monotoninen y-akselin suhteen. Tällöin jos lähdetään monikulmion ylimmästä kulmapisteestä etenemään kumpaa tahansa reunaa pitkin ja edetään alimpaan kulmapisteeseen, ei missään vaiheessa kuljeta ylöspäin.

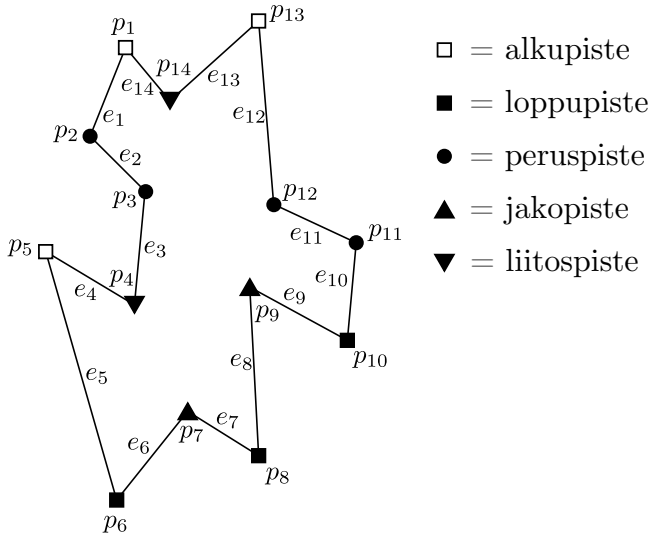
Ajassa $O(n \log n)$ toimivan algoritmin tasoverkon jakamiseen monotonisiin osiin esittelivät Lee ja Preparata [LP77], ja sitä hyödynsivät monikulmioiden kolmioinnissa Garey ja kumppanit [GJPT78]. Tässä esitettävä versio monikulmion jakamisesta monotonisiin osiin perustuu de Bergin ja kumppaneiden [dBvKOS00] näkemykseen Leen ja Preparatan algoritmista.

Oletetaan, että kaikilla monikulmion kulmapisteillä on eri y-koordinaatti. Tämä vaatimus ei todellisuudessa tuota ongelmia, sillä voimme kuvitella monikulmiota kierretyksi minimaalisesti myötäpäivään, jolloin kahdesta samalla korkeudella olevasta pisteestä ajatellaan vasemmanpuoleisen olevan ylempänä. Kuvitelkaamme polkua monikulmion ylimmästä kulmapisteestä alimpaan joko vasenta tai oikeaa reunaa pitkin. Kulmapisteitä, joissa polku vaihtaa suuntaa alaspäin kulkevasta ylöspäin kulkevaksi tai toisin päin, kutsumme *käännöspisteiksi*. Ongelmana on hankkiutua monotonisuuden rikkovista käännöspisteistä eroon.

Luokittelemme monikulmion kulmapisteet viiteen tyyppiin kuten kuvassa 2. Näistä neljä ovat käännöspisteitä: Kulmapiste p on *alkupiste*, jos sen kummatkin naapurit sijaitsevat sen alapuolella ja monikulmion sisus on sen alapuolella. Vastaavasti jos monikulmion sisus on p:n yläpuolella, on p *jakopiste*. Samaan tapaan määritellään *loppupiste* ja *liitospiste* vaihtamalla yläpuoli ja alapuoli toisin päin. Viidennen tyypin pisteet eli kulmapisteet, jotka eivät ole käännöspisteitä, ovat *peruspisteitä*. Monikulmio on y-monotoninen, jos siinä ei yhtään jako- eikä liitospistettä. Tarkoituksena on päästä näistä eroon lisäämällä monikulmioon jakopisteistä ylöspäin ja vastaavasti liitospisteistä alaspäin kulkevia lävistäjiä.

Algoritmi perustuu tasoa pitkin ylhäältä alas kulkevaan pyyhkäisyviivaan, joka pysähtyy kunkin kulmapisteen kohdalla. Olkoon nyt $p_1, p_2, \ldots, p_n$ lista monikulmion kulmapisteistä reunaviivan mukaan vastapäivään lueteltuna. Merkitään vastaavia reunajanoja listalla $e_1, e_2, \ldots, e_n$, missä $e_i = \overline{v_i v_{i+1}}$, kun $1 \leq i < n$, ja $e_n = \overline{v_n v_1}$.

Oletetaan, että pyyhkäisyviiva on jakopisteen p_i kohdalla. Olkoon e_j reunajana,



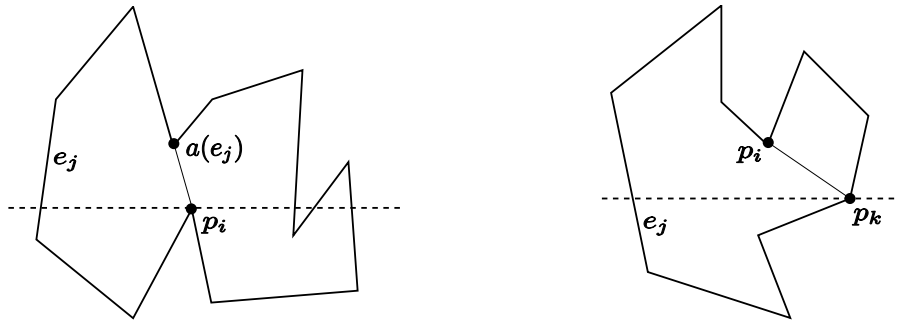
Kuva 2: Kulmapisteiden luokittely viiteen tyyppiin. (Pohjautuu de Bergin ja kumppaneiden [dBvKOS00] kuvaan 3.3.)

joka kohdataan ensimmäisenä kun lähdetään kulkemaan pisteestä p_i pyyhkäisyviivaa pitkin vasemmalle. Käytämme merkintää $a(e_j)$ viittaamaan alimpaan sellaiseen pyyhkäisyviivan yläpuolella ja janan e_j oikealla puolella olevaan pisteeseen, josta vaakatasossa vasemmalle janaan e_j piirretty jana kulkee kokonaan monikulmion sisällä. Jos tällaista pistettä ei ole, viittaa $a(e_j)$ janan e_j ylempään päätepisteeseen. Nyt voidaan vetää lävistäjän pisteiden $a(e_j)$ ja p_i välille (kuva 3a).

Vastaavasti halutaan yhdistää liitospisteet alapuolella olevaan pisteeseen. Olkoon p_i liitospiste ja e_j sen vasemmalla puolella oleva reunajana. Kun pyyhkäisyviiva kohtaa pisteen p_i, tulee siitä $a(e_j)$. Olkoon nyt p_k piste, joka myöhemmin korvaa pisteen p_i janan e_j alimpana pisteenä $a(e_j)$. Tässä tapauksessa pisteet p_i ja p_k voidaan yhdistää lävistäjällä (kuva 3b). Jos tällaista pistettä p_k ei ole, vaan p_i on janan e_j viimeinen alin piste $a(e_j)$, voidaan p_i yhdistää lävistäjällä e_j:n alempaan päätepisteeseen.

Kulmapisteiden ylhäältä alas läpikäymiseksi sijoitetaan ne y-koordinaatin mukaiseen maksimikekoon S. Huipulle tulee siis monikulmion ylin kulmapiste. Algoritmin aikana täytyy pystyä selvittämään kohdattujen pisteiden vasemmalla puolella olevia reunajanoja. Tähän tarkoitukseen ylläpidetään binäärihakupuuta $\mathcal{R}$, joka sisältää kullakin hetkellä pyyhkäisyviivan leikkaamat vasemmanpuoleiset reunajanat vasemmalta oikealle järjestettynä. Lisäksi kunkin puussa olevan janan e_j yhteydessä pidetään tietoa pisteestä $a(e_j)$.

Algoritmi toimii nyt seuraavasti: Aluksi muodostetaan kulmapistekeko S ja aluste-



(a) (b)

Kuva 3: Lävistäjien lisääminen jako- ja liitospisteiden yhteyteen: (a) Jakopiste p_i ja alin sen yläpuolella oleva piste $a(e_j)$ voidaan yhdistää lävistäjällä. (b) Edellinen alin piste p_i, joka on liitospiste voidaan yhdistää lävistäjällä tulevaan liitospisteeseen p_k. (Pohjautuu de Bergin ja kumppaneiden [dBvKOS00] kuviin.)

taan tyhjä janojen hakupuu $\mathcal{R}$. Tämän jälkeen poimitaan kulmapisteet yksitellen keosta S, ja riippuen poimitun pisteen p_i tyypistä suoritetaan jokin seuraavista:

- Jos p_i on *alkupiste*, lisätään e_i puuhun $\mathcal{R}$ ja asetetaan $a(e_i) \leftarrow p_i$. Kuvan 2 tapauksessa lisäämme alkupisteessä p_1 reunajanan e_1 puuhun.

- Jos p_i on *loppupiste*, tarkistetaan onko $a(e_{i-1})$ liitospiste ja myönteisessä tapauksessa lisätään lävistäjä sen ja pisteen p_i välille. Lopuksi poistetaan e_{i-1} puusta $\mathcal{R}$. Esimerkin pisteessä p_{10} on sitä vastaava alin piste $a(e_9) = p_9$. Koska p_9 ei ole liitospiste, lävistäjää ei tarvitse lisätä.

- Jos p_i on *jakopiste*, etsitään puusta $\mathcal{R}$ lähin vasemmanpuoleinen jana e_j ja lisätään lävistäjä pisteiden p_i ja $a(e_j)$ välille. Tämän jälkeen lisätään e_i puuhun $\mathcal{R}$ sekä asetetaan $a(e_j) \leftarrow p_i$ ja $a(e_i) \leftarrow p_i$. Esimerkkikuvan pisteessä p_7 on vasemmanpuoleinen reunajana e_5. Vastaava alin piste on $a(e_5) = p_4$, joten lisäämme lävistäjän pisteiden p_7 ja p_4 välille.

- Jos p_i on *liitospiste*, tarkistetaan, onko $a(e_{i-1})$ liitospiste, ja myönteisessä tapauksessa lisätään lävistäjä sen ja pisteen p_i välille. Sitten poistetaan e_{i-1} puusta $\mathcal{R}$, ja etsitään pistettä p_i lähin vasemmanpuoleinen jana e_j. Jos $a(e_j)$ on liitospiste, lisätään lävistäjä sen ja pisteen p_i välille. Lopuksi asetetaan $a(e_j) \leftarrow p_i$. Esimerkissä pisteen p_4 kohdalla vasemmanpuoleinen reunajana on e_5. Kumpikaan pisteistä $a(e_3) = p_9$ ja $a(e_5) = p_5$ ei ole liitospiste, joten lävistäjiä ei tarvitse lisätä.

- Jos p_i on *peruspiste* ja monikulmion sisus on sen *oikealla* puolella, tarkistetaan onko $a(e_{i-1})$ liitospiste ja myönteisessä tapauksessa lisätään lävistäjä sen ja pisteen p_i välille. Tämän jälkeen poistetaan e_{i-1} puusta $\mathcal{R}$, lisätään e_i puuhun $\mathcal{R}$ ja asetetaan $a(e_i) \leftarrow p_i$. Esimerkin pisteessä p_2 on $a(e_1) = p_{14}$ liitospiste, joten lisätään lävistäjä pisteestä p_2 pisteeseen p_{14}.

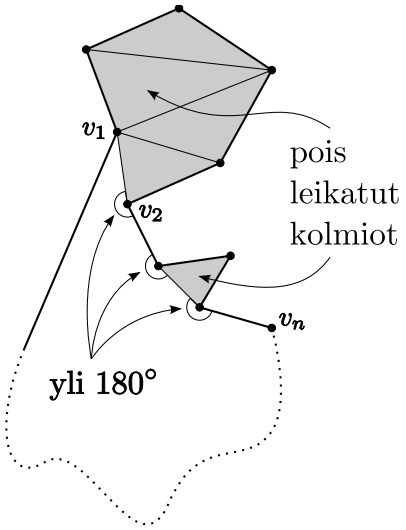
- Jos p_i on *peruspiste* ja monikulmion sisus on sen *vasemmalla* puolella, etsitään puusta $\mathcal{R}$ lähin vasemmanpuoleinen jana e_j. Jos $a(e_i)$ on liitospiste, lisätään lävistäjä sen ja pisteen p_i välille. Lopuksi asetetaan $a(e_j) \leftarrow p_i$. Esimerkkikuvan pisteessä p_{12} on vasemmanpuoleinen reunajana e_3. Koska $a(e_3) = p_3$ ei ole liitospiste, ei lävistäjää tarvitse lisätä.

Kun kaikki pisteet on käsitelty, ovat muodostuneet osat monotonisia. Läpikäyntiä varten kulmapisteet täytyy aluksi järjestää y-koordinaatin mukaan ylhäältä alas. Tämä sujuu ajassa $O(n \log n)$. Keon S muodostaminen onnistuu lineaarisessa ajassa, ja hakupuun $\mathcal{R}$ alustaminen vie vakioajan. Pisteen poimiminen keosta S ja kaikki puulle $\mathcal{R}$ tehtävät operaatiot vievät ajan $O(\log n)$. Näitä tehdään korkeintaan vakiomäärä kierrosta kohden, joten yhden pisteen käsittelyn aikavaativuus on $O(\log n)$. Pisteitä täytyy käsitellä n kappaletta, joten koko algoritmi suoriutuu siis ajassa $O(n \log n)$.

2.2 Monotonisen monikulmion kolmiointi

Osaamme nyt jakaa monikulmion monotonisiin alimonikulmioihin $O(n \log n)$-ajassa, ja jäljellä on näiden alimonikulmioiden kolmioiminen. Lineaarisen algoritmin monotonisen monikulmioin kolmiointiin esittelivät ensimmäisen kerran Garey ja kumppanit [GJPT78]. Esitämme version, joka perustuu sekä kyseiseen Gareyn ja kumppaneiden artikkeliin että de Bergin ja kumppaneiden [dBvKOS00] kuvaukseen samasta algoritmista. Olkoon $P = \{p_1, p_2, \ldots, p_n\}$ *aidosti* y-monotoninen monikulmio, jonka kulmapisteet on lueteltu järjestyksessä. Aidolla y-monotonisuudella tarkoitamme, että P ei sisällä vaakasuoria reunaviivoja. Tätä voimme käytännössä simuloida kuten aiemminkin, ajattelemalla samalla korkeudella olevista pisteistä aina vasemmanpuoleinen ylemmäksi.

Yhdistetään aluksi P:n vasemman ja oikean laidan kulmapisteet y-koordinaatin mukaan laskevaan järjestykseen. Mikäli kahdella pisteellä on sama y-koordinaatti, tulee ensin siis vasemmanpuoleinen. Järjestäminen onnistuu lineaarisessa ajassa, sil-



Kuva 4: Jäljellä olevan monikulmion käsittelemätön osa on suppilon muotoinen. (Pohjautuu de Bergin ja kumppaneiden [dBvKOS00] kuvaan.)

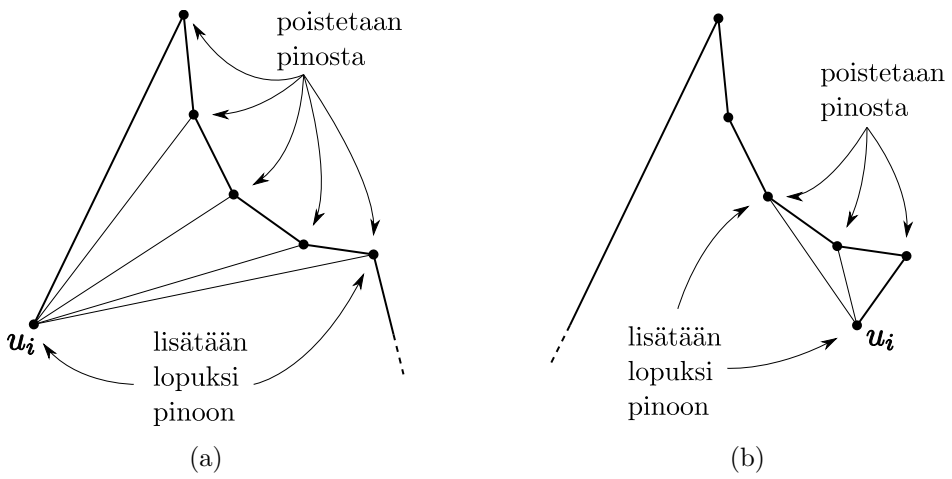
lä se sisältää vain kahden jo järjestyksessä olevan listan yhdistämisen. Olkoon nyt $u_1, u_2, \ldots, u_n$ tämä järjestetty lista kulmapisteitä.

Käsittelemme kulmapisteet $u_1, u_2, \ldots, u_n$ järjestyksessä ylhäältä alas, ja jokaisen kulmapisteen käsittelyn yhteydessä lisäämme mahdollisesti yhden tai useamman lävistäjän monikulmioon. Kukin lisätty lävistäjä leikkaa yhden kolmion — ja sen mukana yhden kulmapisteen — pois monikulmiosta.

Tarvitsemme lisäksi kulmapisteiden väliaikaiseen säilytykseen pinoa S. Pino S sisältää kullakin hetkellä ne käsitellyt kulmapisteet, joita ei ole vielä leikattu pois monikulmioista. Kulmapisteet ovat pinossa käsittelyjärjestyksessä, päällimmäisenä on näistä viimeisimmäksi kohdattu eli alin kulmapiste.

Olkoon pinon S sisältö $v_1, v_2, \ldots, v_k$, missä v_k on pinon päällä. Koko algoritmin ajan seuraavaksi käsiteltävän kulmapisteen yläpuolinen osa jäljellä olevasta monikulmiosta on eräänlaisen ylösalaisin käännetyn suppilon muotoinen; toinen laita koostuu pisteestä v_1 alaspäin lähtevän reunajanan yläosasta, ja toinen laita kulmapisteiden $v_1, v_2, \ldots, v_k$ muodostamasta ketjusta. Lisäksi näistä pisteiden $v_2, \ldots, v_{k-1}$ monikulmion sisuksen puoleinen kulma on vähintään 180° (kuva 4). Seuraavaksi käsiteltävä piste on joko samalla laidalla kuin pinon S ketju, eli seuraava v_k:n jälkeen, tai toisella laidalla, eli seuraava v_1:n jälkeen.

Algoritmi toimii nyt seuraavalla tavalla:



Kuva 5: Pisteen u_i käsittelyn yhteydessä suoritettava kolmiointi: (a) Piste u_i on *eri* laidalla kuin pinon päällimmäinen piste. (b) Piste u_i on *samalla* laidalla kuin pinon päällimmäinen piste. (Pohjautuu de Bergin ja kumppaneiden [dBvKOS00] kuviin.)

1. Alustetaan tyhjä pino S ja lisätään kaksi ensimmäistä kulmapistettä u_1 ja u_2 pinoon.

2. Käydään läpi järjestyksessä kulmapisteet $u_2, \ldots, u_{n-1}$. Olkoon kullakin kierroksella u_i käsiteltävä kulmapiste ja $v_1, v_2, \ldots, v_k$ pinon S sisältö kierroksen alussa.

   (a) Jos u_i on *eri* laidalla kuin pinon S päällimmäinen kulmapiste v_k, tyhjennetään pino S ja lisätään lävistäjät kulmapisteestä u_i pisteisiin $v_2, \ldots, v_k$. Lopuksi lisätään v_k ja u_i pinoon (kuva 5a).

   (b) Jos u_i on *samalla* laidalla kuin pinon S päällimmäinen kulmapiste v_k, poistetaan pinon päällimmäisin kulmapiste. Tämän jälkeen poimitaan pinon päältä pisteitä ja lisätään lisätään lävistäjiä pisteestä u_i poimittuihin pisteisiin niin kauan, kun lävistäjät pysyvät monikulmion sisäpuolella ja pinossa riittää pisteitä. Lopuksi lisätään pinosta viimeiseksi poistettu piste (siis joko aluksi poistettu v_k tai viimeisin tämän jälkeen poimittu) ja u_i pinon päälle (kuva 5b).

3. Kun muut kulmapisteet on käsitelty, vedetään lävistäjät viimeisestä kulmapisteestä pinon pisteisiin $v_2, \ldots, v_{k-1}$.

Ensimmäinen askel eli pinon alustaminen vie vakioajan. Kierroksia suoritetaan $n - 3$ kappaletta, ja yksi kierros voi viedä lineaarisen ajan. Kuitenkin koska kullakin kier-

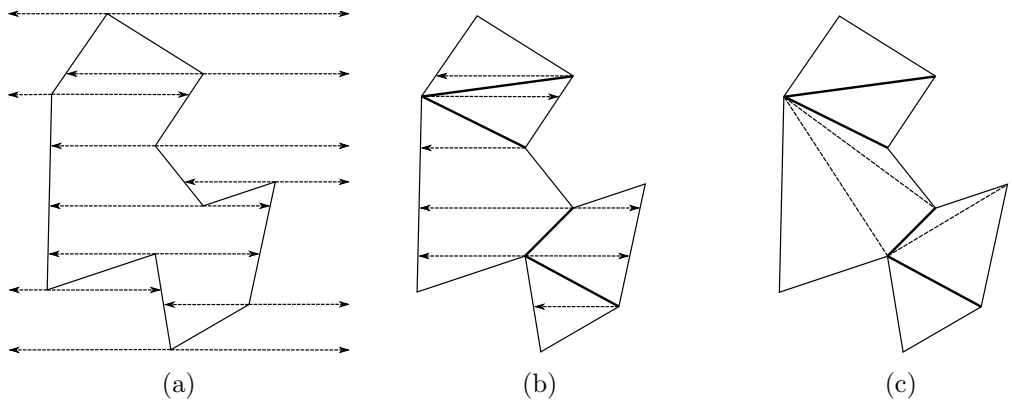
roksella lisätään korkeintaan kaksi kulmapistettä pinoon, tulee lisäyksiä yhteensä enintään $O(n)$. Näin ollen pinosta poistojakin täytyy olla korkeintaan $O(n)$, joten silmukka ja siten koko algoritmi suoriutuu lineaarisessa ajassa.

Olemme nyt esittäneet monikulmion kolmiointiin menetelmän, jonka aikavaativuus on $O(n \log n)$. Menetelmä jakautui kahteen vaiheeseen; ensin jaetaan monikulmio monotonisiksi alimonikulmioiksi ja sen jälkeen kolmioidaan lineaarisessa ajassa syntyneet alimonikulmiot. Tämä algoritmi toimii itse asiassa myös reikiä sisältävillä monikulmioilla, joille saavutettu aikavaativuus $O(n \log n)$ on yleisessä tapauksessa optimaalinen [dBvKOS00]. Seuraavaksi esittelemme uuden välivaiheen, jolla yksinkertaisten monikulmioiden kolmiointia voidaan kuitenkin vielä nopeuttaa.

3 Puolisuunnikashajotelma

Aikavaativuusluokkaa $O(n \log n)$ nopeammat kolmiointialgoritmit perustuvat tyypillisesti *puolisuunnikashajotelman* tai sen muunnelman rakentamiseen kolmioitavasta monikulmiosta [Sei91, Cha91, KKT92, AGR01]. Periaatteena on jakaa monikulmion sisäosa puolisuunnikkaan muotoisiin osiin. Kun jako osiin on muodostettu, on sen perusteella helppo jakaa monikulmio monotonisiin osiin ja edelleen laskea lopullinen kolmiointi. Kaikki tästä eteenpäin esittämämme algoritmit ovatkin erilaisia menetelmiä puolisuunnikashajotelman laskemiseen. Ennen näitä tutustumme kuitenkin tarkemmin puolisuunnikashajotelmaan ja siihen, miten sen avulla voidaan jakaa monikulmio monotonisiin osiin.

Määritellään puolisuunnikashajotelma (trapezoidal decomposition, visibility map) seuraavalla tavalla [Sei91]: Oletetaan, että monikulmion reuna muodostuu joukosta $S = \{s_1, s_2, \ldots, s_n\}$ toisiaan leikkaamattomia ei-vaakasuoria janoja. Lähetetään jokaisesta janan päätepisteestä vaakasuorat säteet sekä vasemmalle että oikealle ja jatketaan jokaista sädettä, kunnes se kohtaa ensimmäisen janan joukosta S. Säteet ja janat jakavat tason osiin, jotka ovat (mahdollisesti toisesta tai kummastakin päästä avoimen) puolisuunnikkaan muotoisia (kuva 6a). Kolmion muotoiset osat ajatellaan puolisuunnikkaiksi, joiden ylä- tai alareunan pituus on nolla. Jos lisäksi oletetaan, että eri kulmapisteillä ei voi olla samaa y-koordinaattia (kuten aiemminkin, tämä rajoitus on helppo kiertää toteutuksen yhteydessä), koskettaa kukin puolisuunnikas enintään kahta naapuria yläpuolella ja enintään kahta naapuria alapuolella. Määritellään puolisuunnikashajotelma $\mathcal{T}(S)$ tarkoittamaan syntyneiden puolisuunnikkaiden joukkoa. Lisäksi yleistetään määritelmä koskemaan mitä tahansa toisiaan leik-



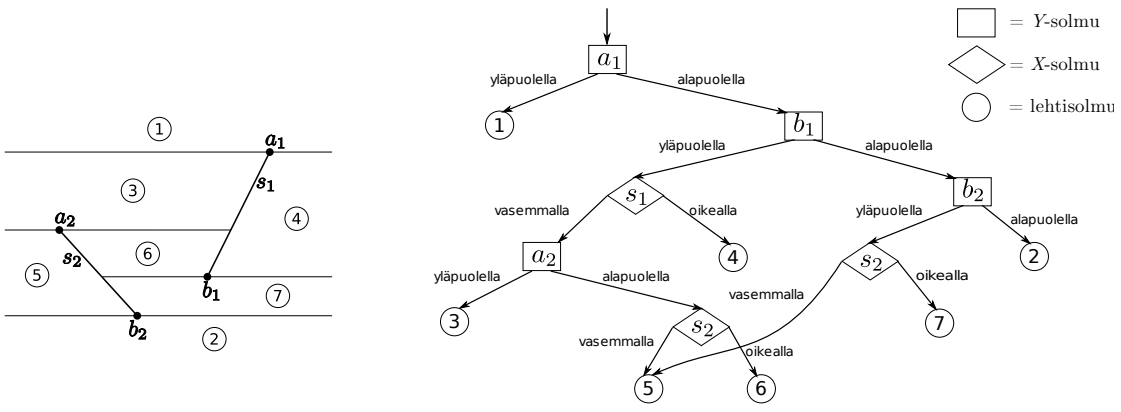
Kuva 6: Puolisuunnikashajotelman kolmiointi: (a) Alkuperäinen puolisuunnikashajotelma. (b) Ulkopuolelle jäävät puolisuunnikkaat poistettu ja diagonaalit lisätty. (c) Lopullinen y-monotonisten alimonikulmioiden kolmiointi. (Pohjautuu Seidelin [Sei91] kuvaan 3.)

kaamattomien janojen joukkoa ja näin ollen erityisesti mitä tahansa monikulmion reunaviivan muodostavien janojen osajoukkoa $S' \subset S$.

Monikulmio P voidaan siitä muodostetun puolisuunnikashajotelman $\mathcal{T}(S)$ avulla kolmioida ajassa $O(n)$ seuraavasti: Aluksi poistetaan hajotelmasta $\mathcal{T}(S)$ monikulmion ulkopuolelle jäävät osat. Kustakin jäljelle jääneestä puolisuunnikkaasta tarkastetaan, sisältävätkö sen sivut kaksi kulmapistettä, jotka eivät kuitenkaan sijaitse samalla sivulla. Myönteisessä tapauksessa pisteiden välille vedetään lävistäjä (kuva 6b). Näin lisätyt lävistäjä jakavat P:n monikulmioihin, joiden toinen laita koostuu yhdestä janasta ja toinen y-monotonisesta janaketjusta. Syntyneet alimonikulmiot ovat siis y-monotonisia ja ne voidaan näin ollen kolmioida edellä mainitulla tavalla lineaarisessa ajassa (kuva 6c). Ongelmaksi jää puolisuunnikashajotelman muodostaminen tarpeeksi nopeasti.

4 Nopea satunnaisalgoritmi kolmiointiin

Esittelemme puolisuunnikashajotelman laskemiseen Seidelin julkaiseman algoritmin [Sei91], jonka odotusarvoinen suoritusaika on $O(n \log^* n)$. Aluksi kuvaamme yksinkertaistetun, odotusarvoisesti ajassa $O(n \log n)$ suoriutuvan version. Tämän jälkeen muunnamme algoritmia hieman, jotta pääsemme haluuttuun luokan $O(n \log^* n)$ aikavaativuuteen.



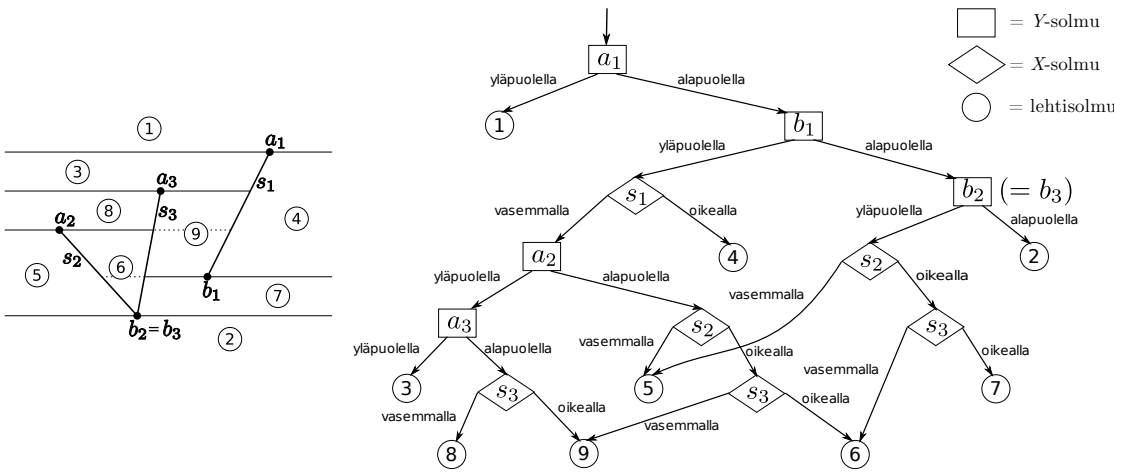
(a) (b)

Kuva 7: (a) Kahdesta janasta muodostettu puolisuunnikashajotelma. (b) Vastaava hakurakenne. (Pohjautuu Seidelin [Sei91] kuvaan 4.)

4.1 Yksinkertainen $O(n \log n)$-aikainen algoritmi

Olkoon S monikulmion reunajanojen joukko kuten edellä. Tallennamme janajoukon S puolisuunnikashajotelman $\mathcal{T}(S)$ tietorakenteeseen, josta jokaiselle puolisuunnikkaalle $\tau \in \mathcal{T}(S)$ voidaan vakioajassa selvittää sekä sitä oikealta ja vasemmalta rajoittavat janat että ylä- ja alapuoliset naapuripuolisuunnikkaat. Määritellään lisäksi pisteen sijainnin hakurakenne $\mathcal{Q}(S)$ hajotelmalle $\mathcal{T}(S)$ olemaan seuraavanlainen suunnattu syklitön verkko: Verkon $\mathcal{Q}(S)$ solmut jakautuvat päätössolmuiksi ja lehtisolmuiksi, joista jälkimmäisiä on täsmälleen yksi jokaista joukon $\mathcal{T}(S)$ puolisuunnikasta kohti. Päätössolmut ovat joko X- tai Y-solmuja ja kummassakin tapauksessa niistä lähtee aina kaksi kaarta. Päätössolmuista X-solmut vastaavat joukon S janoja, ja Y-solmut vastaavat janojen päätepisteiden y-koordinaatteja (tai päätepisteistä lähetettyjä säteitä). Yhteen päätössolmuun ei tule kaaria, ja se toimii lähtösolmuna. Verkko $\mathcal{Q}(S)$ muistuttaa siis binäärihakupuuta: lähtösolmu vastaa juurisolmua ja lehtisolmut lehtiä. Binääripuista poiketen solmuilla voi kuitenkin olla useita (jopa eri tasoilla olevia) vanhempia. Esimerkki kahden janan puolisuunnikashajotelmasta ja sitä vastaavasta hakurakenteesta on kuvassa 7.

Kun tiedetään tason pisteen p koordinaatit, voidaan hakurakenteen $\mathcal{Q}(S)$ avulla selvittää pisteen p sisältävä puolisuunnikas $\tau_p \in \mathcal{T}(S)$ seuraavalla tavalla: Lähdetään liikkeelle lähtösolmusta ja edetään suunnatuista kaarista muodostuvaa polkua



Kuva 8: (a) Kuvan 7 tilanteesta uusi jana lisäämällä muodostettu uusi puolisuunnikashajotelma. (b) Vastaava päivitetty hakurakenne. (Pohjautuu Seidelin [Sei91] kuvaan 4.)

pitkin. Kussakin X-solmussa tutkitaan, onko p vasemmalla vai oikealla puolella solmuun liittyvää janaa, ja päätetään sen perusteella, kumpaa lähtevää kaarta seurataan. Vastaavasti Y-solmuissa verrataan y-koordinaattia. Näin päädytään lehtisolmuun, joka vastaa haettua puolisuunnikasta τ_p. Oletetaan lisäksi, että puolisuunnikkaan $\tau \in \mathcal{T}(S)$ perusteella voidaan selvittää vastaava lehtisolmu rakenteesta $\mathcal{Q}(S)$ vakioajassa ja toisin päin.

Algoritmi kokonaisen monikulmion puolisuunnikashajotelman muodostamiseksi on seuraavanlainen: Aloitetaan tyhjästä janajoukosta $S = \emptyset$ ja lisätään monikulmion reunajanat satunnaisessa järjestyksessä yksitellen joukkoon S. Jokaisen lisäyksen yhteydessä päivitetään sekä puolisuunnikashajotelmaa $\mathcal{T}(S)$ että hakurakennetta $\mathcal{Q}(S)$. Olkoon nyt S joukko janoja ja $S' = S \cup \{s\}$ vastaava joukko, johon on lisätty jana $s \notin S$. Haluamme muodostaa hajotelman $\mathcal{T}(S')$ ja hakurakenteen $\mathcal{Q}(S')$, kun meillä on olemassa hajotelma $\mathcal{T}(S)$ ja hakurakenne $\mathcal{Q}(S)$. Kuvassa 8 on esimerkki hajotelmasta ja hakurakenteesta, jotka on saatu lisäämällä uusi jana kuvan 7 tilanteeseen.

Jos janan s ylempi päätepiste a ei ole minkään toisen joukossa S jo olevan janan päätepiste, selvitetään aluksi hakurakenteen $\mathcal{Q}(S)$ avulla puolisuunnikas $\tau_a \in \mathcal{T}(S)$, joka sisältää päätepisteen a. Löydetty puolisuunnikas τ_a jaetaan kahdeksi puolisuunnikkaaksi pisteen a kautta kulkevalla vaakasuoralla säteellä. Lisäksi puolisuunnikasta

τ_a vastaava lehtisolmu hakurakenteessa $\mathcal{Q}(S)$ muutetaan Y-solmuksi, jonka seuraajiksi luodaan kahta uutta puolisuunnikasta vastaavat lehtisolmut. Jos taas a on jo jonkin joukon S janan päätepiste, ei rakenteille tehdä mitään. Samaan tapaan käsitellään myös janan s alempi päätepiste b. Kuvassa 8 ylempää päätepistettä a_3 ei ollut edellisessä vaiheessa, kun taas alempi päätepiste b_3 oli jo aiemmin lisätyn janan päätepiste (b_2).

Seuraavaksi kuljetaan puolisuunnikashajotelman läpi janaa s pitkin ja jaetaan jokainen janan leikkaama puolisuunnikas kahtia. Samalla yhdistetään mahdolliset syntyvät päällekkäiset samojen janojen rajoittamat puolisuunnikkaat. Lisäksi päivitetään hakurakenne lisäämällä uusia puolisuunnikkaita vastaavat lehtisolmut ja muuttamalla halkaistuja puolisuunnikkaita vastaavat lehtisolmut X-solmuiksi. Näin olemme muodostaneet halutut $\mathcal{T}(S')$ ja $\mathcal{Q}(S')$. Kuvan 8 esimerkissä leikkasi uusi jana kolme puolisuunnikasta kahtia, ja syntyneistä kuudesta puolisuunnikkaasta yhdistettiin kummaltakin puolelta janaa kaksi.

Kun järjestys, jossa monikulmion reunajanat lisätään joukkoon S, on satunnainen (jokainen permutaatio on yhtä todennäköinen), kuluu yhden janan päätepisteet sisältävien puolisuunnikkaiden etsimiseen hakurakenteen $\mathcal{Q}(S)$ avulla odotusarvoisesti aikaa $O(\log n)$. Lisäksi puolisuunnikashajotelman läpikulkeminen janaa pitkin sekä puolisuunnikkaiden halkaiseminen kahtia onnistuu tällöin odotusarvoisesti vakioajassa. Näin ollen monikulmion koko puolisuunnikashajotelman muodostamisen odotusarvoinen aikavaatimus on $O(n \log n)$, missä n on monikulmion kulmien määrä.

4.2 $O(n \log^* n)$-aikainen laajennus

Esitellyllä algoritmilla pullonkaulaksi muodostuvaa hakurakenteen $\mathcal{Q}(S)$ käyttöä voidaan nopeuttaa muodostamalla hajotelma useassa syklissä ja laskemalla osittaisia pisteiden sijaintitietoja kullekin syklille. Seuraavassa merkintä $\log^{(i)} n$ tarkoittaa i. kertaluvun iteroitua logaritmia (eli $\log^{(0)} n = n$ ja $\log^{(i)} n = \log(\log^{(i-1)} n)$, kun $i > 0$). Vastaavasti merkitään $\log^* n = \max\{i \in \mathbf{N} | \log^{(i-1)} n \geq 1\}$. Muunnetussa algoritmissa syklejä tarvitaan yhteensä $N = \log^* n + 1$. Kussakin syklissä $i \in \{1, \ldots, N-1\}$ laajennetaan puolisuunnikashajotelmaa, kunnes se sisältää $\lceil n/log^{(i)} n \rceil$ janaa. Viimeisellä syklillä lisätään hajotelmaan jäljellä jääneet janat.

Jokaisen syklin lopussa selvitetään kaikkien vielä jäljellä olevien janojen päätepisteiden sijainnit (puolisuunnikkaat, joissa ne sijaitsevat) sen hetkisessä puolisuunnikashajotelmassa. Pisteiden sijaintien selvittäminen onnistuu yhteensä lineaarisessa

ajassa kulkemalla kaikki monikulmion kulmapisteet läpi järjestyksessä monikulmion reunaviivaa pitkin ja edeten samalla hajotelmassa puolisuunnikkaasta toiseen. Selvitetyt sijainnit tallennetaan pisteiden yhteyteen, ja niitä käytetään seuraavassa syklissä hyödyksi: Nyt pisteen sijaintia etsittäessä ei tarvitse käydä koko polkua läpi hakurakenteessa $\mathcal{Q}(S)$, vaan eteneminen voidaan aloittaa aiemmin selvitettyjä sijainteja vastaavista solmuista.

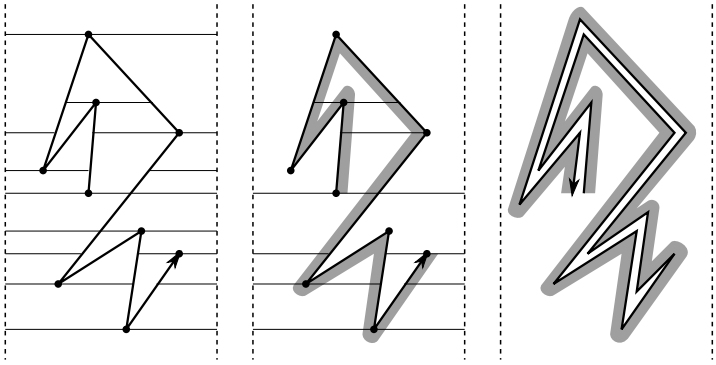
Polun loppuosan läpikäynti syklissä i vaatii ajan $O(log^{(i)}n)$, joten yhden syklin kaikista hauista selvitään n:n suhteen lineaarisessa ajassa. Tällöin siis suoritusaika yhden syklin osalta on $O(n)$. Koska syklejä tehtiin yhteensä $O(\log^* n)$ kappaletta, saadaan algoritmin odotusarvoinen aikavaativuus pudotettua luokkaan $O(n \log^* n)$.

Olemme nyt esitelleet odotusarvoisesti ajassa $O(n \log n)$ suoriutuvan algoritmin monikulmion puolisuunnikashajotelman muodostamiseen sekä muokatun version, jonka odotusarvoinen aikavaativuus on $O(n \log^* n)$. Algoritmi on suhteellisen yksinkertainen ja siten sen avulla on toteutettavissa käytännöllinen algoritmi monikulmion kolmioimiseen [NM95]. Esimerkiksi $\log^* 10000 = 4$, joten 10000-kulmaisen monikulmion tapauksessa $O(n \log^* n)$-aikaisellä algoritmilla syklejä tarvittaisiin viisi. Käytännössä 2-kantainen $\log^*$ ei kasva suuremmaksi kuin 5, joten saavutettua aikavaativuutta voidaan pitää lähes lineaarisena. Toisaalta kun n muuttuu vaikkapa arvosta 10 arvoon 10^9, kasvaa $\log_2 n$ alle 10-kertaiseksi ($3.3 \rightarrow 29.9$) samalla kun 2-kantainen $\log^* n$ hieman yli puolitoistakertaistuu ($3 \rightarrow 5$). Ero hitaampaan $O(n \log n)$-aikaiseen versioonkaan ei siten vielä miljardikulmaisella monikulmiollakaan vaikuta kovin suurelta.

5 Nopea deterministinen kolmiointi

Tässä luvussa esittelemme suhteellisen karkealla tasolla Kirkpatrickin ja kumppaneiden [KKT92] deterministisen $O(n \log \log n)$-aikavaativuuden algoritmin puolisuunnikashajotelman muodostamiseen. Algoritmi käyttää puolisuunnikashajotelman muunnosta, jota kutsumme *pyörähdyshajotelmaksi*, ja perustuu hajota ja hallitse -menetelmään. Ideana on jakaa monikulmion reunapolku sopivan pituisiin osapolkuihin, joille kullekin muodostetaan pyörähdyshajotelma kutsumalla algoritmia rekursiivisesti. Osapolkujen pyörähdyshajotelmien avulla jaetaan taso osiin, joiden sisäosille lasketaan yhdistetyt hajotelmat. Osien hajotelmat yhdistämällä rakennetaan lopuksi koko reunapolun hajotelma.

Jos s on suunnattu reunajana, sanotaan sen kulkusuunnassa oikeaa puolta *positiivi-*



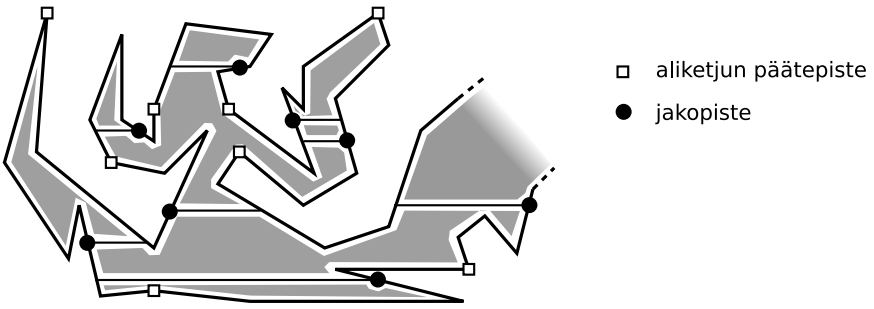
(a) (b) (c)

Kuva 9: Suunnatun reunajanaketjun (a) kaksipuolinen pyörähdyshajotelma, (b) yksipuolinen pyörähdyshajotelma ja (c) kahdennus. Harmaa väri tarkoittaa ketjun positiivista puolta. (Pohjautuu Kirkpatrickin ja kumppaneiden [KKT92] kuvaan 2.)

seksi ja vasenta puolta *negatiiviseksi*. Vastaavasti määritellään suunnatun reunajanaketjun positiivinen ja negatiivinen puoli. Pyörähdyshajotelmaa muodostettaessa ajatellaan janaketju pystysuuntaiseen loputtoman korkean sylinterin ulkopinnalle. Tästä saavutetaan se etu, että monikulmioiden ulkopuolista aluetta voidaan käsitellä samaan tapaan kuin sisäpuolta.

Kaksipuolinen pyörähdyshajotelma määritellään nyt samaan lähes tapaan kuin tavallinen puolisuunnikashajotelma: Kustakin janaketjun kulmapisteestä (päätepisteet mukaanlukien) lähetetään vaakasuorat säteet vasemmalle sekä oikealle ja säteitä jatketaan ensimmäiseen kohdattuun reunajanaan saakka. Aiemmasta poiketen kuitenkin sylinterin reunan kohtaavat säteet pyörähtävätkin sylinterin takapuolelta ympäri, ilmestyvät sylinterin toiselta reunalta takaisin ja jatkavat matkaa ensimmäiseen kohtaamaansa reunaviivaan. Muodostuvien osien ajatellaan samaan tapaan sylinterin reunan kohdattuaan jatkuvan toisella puolella (kuva 9a). Janaketjun *yksipuolinen* pyörähdyshajotelma saadaan ottamalla kaksipuolisesta hajotelmasta mukaan vain janaketjun positiiviselle puolelle lähetetyt säteet. Päätepisteistä otetaan kumpaankin suuntaan lähetetty säde mukaan (kuva 9b).

Suunnattu reunajanaketju voidaan *kahdentaa* kopioimalla se, siirtämällä kopion kulmapisteitä hieman alkuperäisen ketjun negatiiviselle puolelle, vaihtamalla kopion kulkusuunta päinvastaiseksi sekä liittämällä sen alkupiste alkuperäisen ketjun päätepisteeseen. Näin saadun *kahdennuksen* positiivinen puoli kattaa lähes koko sylinterin pinnan lukuunottamatta mielivaltaisen kapeaa aluetta alkuperäisen ketjun ja



Kuva 10: Aliketjuista koostuvan janaketjun positiivisen puolen jako alueisiin jakopisteistä lähtevillä säteillä. (Pohjautuu Kirkpatrickin ja kumppaneiden [KKT92] kuvaan 3.)

sen kopion välissä. Erityisesti huomataan, että ketjun kahdennuksen yksipuolisesta pyörähdyshajotelmasta päästään alkuperäisen ketjun kaksipuoliseen pyörähdyshajotelmaan siirtämällä ketjun kopio takaisin alkuperäisen ketjun päälle. Kahdennusta käytetäänkin nimenomaan siirtymiseen yksipuolisen ja kaksipuolisen pyörähdyshajotelman välillä. Esimerkki kahdennuksesta on kuvassa 9c.

Algoritmi monikulmion kaksipuolisen pyörähdyshajotelman muodostamiseksi toimii nyt pääpiirteittäin seuraavalla tavalla: Aluksi algoritmi saa syötteenä monikulmion reunajanaketjun jostain kulmapisteestä aloitettuna. Olkoon S syötteenä saatu ketju ja $r = |S|$ ketjun pituus (janojen lukumäärä). Ensimmäisessä vaiheessa pätkitään janaketju peräkkäisiksi *aliketjuiksi*, joiden pituus on luokkaa $O(r^{2/3})$. Aliketjuja muodostuu tällöin siis $O(\sqrt[3]{r})$ kappaletta. Kullekin aliketjulle lasketaan kaksipuolinen puolisuunnikashajotelma kutsumalla algoritmia rekursiivisesti. Tämän jälkeen ketju S kahdennetaan ketjuksi S' ja alkuperäisen ketjun S aliketjujen pyörähdyshajotelmista muodostetaan kahdennuksen S' vastaavien aliketjujen pyörähdyshajotelmat. Tässä vaiheessa käytössä on siis kahdennettu ketju S', jonka aliketjuille on tiedossa pyörähdyshajotelmat.

Seuraavaksi kustakin aliketjusta etsitään tietyt ominaisuudet täyttävä piste, joka jakaa ketjun kahteen osaan. Jokaisesta tällaisesta *jakopisteestä* lähetetään yksi vaakasuora säde janaketjun S' positiiviselle puolelle samaan tapaan kuin yksipuolista pyörähdyshajotelmaa muodostettaessa, ja jatketaan sädettä kunnes se törmää johonkin toiseen osaan ketjua S'. Lähetetyt säteet jakavat ketjun S' positiivisen puolen alueisiin (kuva 10). Sopivien jakopisteiden etsiminen, säteiden lähettäminen ja jako alueisiin voidaan aliketjujen pyörähdyshajotelmien avulla tehdä ajassa $O(r)$.

Tämän jälkeen tavoitteena on muodostaa ketjun S' yksipuolinen pyörähdyshajotelma alue kerrallaan. Kutakin aluetta rajaava reunaviiva koostuu nyt ketjun S' pätkistä sekä vaakasuorista jakosäteistä. Osalle alueista — niille, joita rajoittaa korkeintaan kaksi vaakasuoraa jakosädettä — voidaan muodostaa pyörähdyshajotelma lineaarisessa ajassa pilkkomalla ja yhdistämällä alueelle osuvia aliketjujen hajotelmia. Loppujen alueiden kanssa joudutaan toimimaan hieman monimutkaisemmin: Ensin kunkin alueen reunaviiva jaetaan $O(r^{2/3})$ pätkään ja alueelle osuvien aliketjujen hajotelmia pilkkomalla ja yhdistämällä muodostetaan lineaarisessa ajassa näille pätkille kaksipuoliset pyörähdyshajotelmat. Tämän jälkeen olemme kunkin alueen reunaviivan osalta saman tyyppisessä tilanteessa kuin algoritmin alussa rekursioaskeleen ja kahdentamisen jälkeen — tavoitteena on muodostaa alueen sisälle yksipuolinen pyörähdyshajotelma. Alueiden sisäiset pyörähdyshajotelmat voidaan siis rakentaa ajattelemalla kunkin alueen sisäpuoli aluetta rajoittavan reunajanaketjun positiiviseksi puoleksi ja suorittamalla algoritmin loppuosa rekursiivisesti kunkin alueen reunajanaketjulle.

Kun hajotelma on saatu muodostettua kullekin alueelle, yhdistetään nämä palaset kokonaiseksi ketjun S' yksipuoliseksi pyörähdyshajotelmaksi. Koska ketju S' oli ketjun S kahdennus, vastaa muodostettu S':n yksipuolinen hajotelma (pientä siirtoa lukuunottamatta) S:n kaksipuolinen hajotelmaa. Näin olemme saaneet muodostettua S:n kaksipuolisen hajotelman. Algoritmin yksi edellä kuvatun kaltainen kierros vie alun aliketjuja koskevaa rekursiota lukuunottamatta lineaarisen ajan syötteenä saadun ketjun pituuden suhteen. Tästä seuraa, että algoritmin aikavaativuus on $O(n \log \log n)$.

Olemme esitelleet aikavaativuuden $O(n \log \log n)$ deterministisen algoritmin monikulmion pyörähdyshajotelman muodostamiseen. Algoritmista on mahdollista muokata myös versio, joka laskee pyörähdyshajotelman ajassa $O(n \log^* n)$ monikulmioille, joiden kulmapisteillä on kokonaislukukoordinaatit. Koko monikulmion kaksipuolinen pyörähdyshajotelma vastaa monikulmion sisäpuolella täysin tavallista puolisuunnikashajotelmaa, joten kun se on muodostettu, voidaan monikulmio kolmioida lineaarisessa ajassa aiemmin mainitulla menetelmällä. Esitetty algoritmi on kuitenkin monimutkainen ja hankalasti toteutettava verrattuna edellä esitettyihin aikavaativuuden $O(n \log n)$ deterministiseen algoritmiin ja $O(n \log^* n)$-ajan satunnaisalgoritmiin, joten sen käytännön merkitystä voidaan pitää kyseenalaisena.

6 Kolmiointi lineaarisessa ajassa

Edellä olemme esitelleet yksinkertaisen monikulmion kolmioimiseen lähelle lineaarista nopeutta pääseviä algoritmeja. Chazelle [Cha91] osoitti, että kolmiointi on mahdollista suorittaa myös lineaarisessa ajassa. Tässä tutustumme lyhyesti sekä Chazellen deterministiseen algoritmiin että Amaton ja kumppaneiden [AGR01] satunnaisalgoritmiin, jotka kumpikin toimivat lineaarisessa ajassa. Kuten kaksi edellistäkin algoritmia, myös nämä perustuvat puolisuunnikashajotelman laskemiseen kolmioitavalle monikulmiolle. Kummastakin esitämme vain korkean tason hahmotelma algoritmin toiminnasta.

6.1 Deterministinen algoritmi

Kuten edellä mainitsimme, Chazellen [Cha91] deterministinen algoritmi perustuu puolisuunnikashajotelman laskemiseen kolmioitavalle monikulmiolle. Tässä tapauksessa monikulmio ajatellaan pallon pinnalle, jolloin — samoin kuin edellisessä kohdan 5 algoritmissa sylinterin tapauksessa — puolisuunnikashajotelmaa muodostettaessa kulmapisteistä lähetettävät vaakasuorat säteet monikulmion lähettyviltä pois pyrkiessään kierähtävät pallon ympäri ja ilmestyvät toiselta puolelta takaisin.

Kuten kohdassa 5 esitetyssä algoritmissa, myös Chazellen versiossa on keskeisenä ajatuksena muodostaa hajotelma ensin lyhyille monikulmion reunajanaketjun pätkille, ja pätkien hajotelmia yhdistämällä rakentaa yhdistettyjen pidempien pätkien hajotelmia ja lopulta koko reunan (ja siis monikulmion) hajotelma. Normaalisti pätkien hajotelmien yhdistämiseen kuluisi lineaarinen aika, jolloin useita rekursioaskeleita suoritettaessa aikavaativuus kasvaa yli lineaarisen. Merkittävänä erona koko prosessi jaetaankin kahteen vaiheeseen: yhdistämisvaiheeseen, jossa täydellisten hajotelmien sijaan yhdistetäänkin eräänlaisia approksimaatioita niistä, sekä tarkennusvaiheeseen, jossa edellisestä vaiheesta saatu approksimaatiohajotelma tarkennetaan lopulliseksi hajotelmaksi.

Ensin tehtävässä yhdistämisvaiheessa yhdistettävät approksimaatiohajotelmat ovat käytännössä hajotelmia, joista on poistettu osa vaakasuorista säteistä. Kun approksimaatiohajotelmat pidetään algoritmin edetessä sopivan kokoisina (niin, että ne sisältävät tarpeeksi vähän säteitä) ja tietyt muut ehdot täyttävinä, saadaan kahden hajotelman yhdistäminen tehtyä sublineaarisessa ajassa. Tällöin koko yhdistämisvaihe kuluttaa vain lineaarisen ajan.

Lopuksi suoritettava tarkennusvaihe etenee vastakkaiseen suuntaan kuin yhdistämisvaihe koko monikulmion reunaketjusta yhä pienempiin osaketjuihin ja tarkentaa matkan varrella hajotelmaa vaiheittain lisäämällä puuttuvia säteitä päätyen lopulta monikulmion täydelliseen hajotelmaan. Myös tarkennusvaihe saadaan tehtyä lineaarisessa ajassa, joten algoritmin aikavaativuus on $O(n)$.

6.2 Satunnaisalgoritmi

Amaton ja kumppaneiden [AGR01] lineaarinen satunnaisalgoritmi puolisuunnikashajotelman laskemiseksi voidaan nähdä eräänlaisena yhdistelmänä kohdassa 4 esitetystä $O(n \log^* n)$-ajan satunnaisalgoritmista ja edellisen kohdan 6.1 lineaarisesta algoritmista. Kuten kohdan 4 versiossa, myös Amaton ja kumppaneiden algoritmissa hajotelma rakennetaan $O(\log^* n)$ syklissä täydentämällä aina edellisessä syklillä saavutettua osahajotelmaa satunnaisesti valitulla otoksella jäljellä olevista reunaviivoista. Monikulmion reunajanoja ei kuitenkaan käsitellä yksitellen, vaan vaihtelevan pituisina reunan osaketjuina (pituus riippuu syklistä). Kussakin syklissä hajotelmaan lisätään siis satunnaisesti valittu joukko osaketjuja yksittäisten janojen sijaan.

Toisaalta kuten kohdan 6.1 algoritmi, myös tämä jakautuu kahteen vaiheeseen: karkean approksimaation rakentamiseen puolisuunnikashajotelmasta sekä approksimaation tarkentamiseen lopulliseksi puolisuunnikashajotelmaksi. Syklien sisällä approksimaatiohajotelmaa ei kuitenkaan lasketa koko monikulmion reunajanalle vaan edellä mainitulle osaketjujen satunnaisotannalle.

Kehitetyt lineaarisen ajan kolmiointialgoritmit ovat suhteellisen monimutkaisia. Erityisesti kohdassa 6.1 esitettyä Chazellen algoritmia on monimutkaisuutensa vuoksi arvosteltu mahdottomaksi toteuttaa onnistuneesti [Hel01]. Vaikka kohdan 6.2 satunnaisalgoritmi on huomattavasti kohdan 6.1 determinististä algoritmia yksinkertaisempi, on se kuitenkin vastaavasti selvästi monimutkaisempi kuin kohdassa 4 esitetty epäoptimaalinen satunnaisalgoritmi. Tästä syystä silläkään ei liene merkitystä käytännön sovelluksissa [AGR01].

7 Yhteenveto

Monikulmion kolmioinnille on olemassa useita käyttötarkoituksia. Tutustuimme erilaisiin menetelmiin yksinkertaisen monikulmion kolmioimiseen. Aluksi esittelimme

$O(n \log n)$-aikaisen algoritmin yksinkertaisen monikulmion jakamiseen monotonisiin osiin, sekä lineaarisen algoritmin monotonisten monikulmioiden kolmiointiin. Nämä yhdistämällä saimme yksinkertaisen deterministisen menetelmän monikulmion kolmiointiin. Seuraavaksi tutustuimme monikulmion puolisuunnikashajotelmaan ja monotonisiin osiin jakamiseen sen avulla. Tämän jälkeen kävimme läpi $O(n \log^* n)$-aikavaativuuden satunnaisalgoritmin sekä $O(n \log \log n)$-ajan vaativan deterministisen algoritmin puolisuunnikashajotelman muodostamiseen. Lopuksi tutustuimme lyhyesti kahteen algoritmiin, jotka muodostavat puolisuunnikashajotelman lineaarisessa ajassa.

Esitellyistä algoritmeista kaksi ensimmäistä, kohdassa 2 esitelty $O(n \log n)$-aikainen algoritmi sekä kohdassa 4 esitelty $O(n \log^* n)$-ajan satunnaisalgoritmi ovat kohtuullisen nopeita, mutta siitä huolimatta suhteellisen yksinkertaisia ja siten toteutettavissa olevia. Sen sijaan näiden jälkeen kohdassa 5 läpikäyty $O(n \log \log n)$-ajan algoritmi, kohdan 6.2 lineaarinen satunnaisalgoritmi sekä erityisesti kohdan 6.1 lineaarinen algoritmi ovat sen verran monimutkaisia ja hankalasti toteutettavia, että alun yksinkertaisemmat mutta asymptoottisesti hieman hitaammat algoritmit lienevät todellisuudessa nopeampia kaikilla käytännöllisen kokoisilla monikulmioilla. Tosielämän sovelluksissa esitettyjen teoreettisesti nopeiden kolmiointialgoritmien sijaan saatetaan kuitenkin käyttää periaatteessa hitaampia, esimerkiksi neliöllisen ajan vaativia algoritmeja [Hel01]. Käytännön kolmiointiongelmissa nämä voivatkin osoittautua vähintään yhtä nopeiksi, ellei jopa nopeammiksi kuin tässä esitetyt algoritmit.

Lähteet

AGR01 Amato, N. M., Goodrich, M. T. ja Ramos, E. A., A randomized algorithm for triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 26,2(2001), sivut 245–265.

Cha91 Chazelle, B., Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6,5(1991), sivut 485–524.

CTVW89 Clarkson, K. L., Tarjan, R. E. ja Van Wyk, C. J., A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete & Computational Geometry*, 4,5(1989), sivut 423–432.

dBvKOS00 de Berg, M., van Kreveld, M., Overmars, M. ja Schwarzkopf, O., *Com-*

putational Geometry: Algorithms and Applications. Springer, toinen painos, 2000.

GHL+87 Guibas, L., Hershberger, J., Leven, D., Sharir, M. ja Tarjan, R. E., Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2,1(1987), sivut 209–233.

GJPT78 Garey, M. R., Johnson, D. S., Preparata, F. P. ja Tarjan, R. E., Triangulating a simple polygon. *Information Processing Letters*, 7,4(1978), sivut 175–179.

Hel01 Held, M., FIST: Fast industrial-strength triangulation of polygons. *Algorithmica*, 30,4(2001), sivut 563–596.

Jos94 Joseph, O., *Computational Geometry in C.* Cambridge University Press, 1994.

KKT92 Kirkpatrick, D. G., Klawe, M. M. ja Tarjan, R. E., Polygon triangulation in $O(n \log \log n)$ time with simple data structures. *Discrete & Computational Geometry*, 7,4(1992), sivut 329–346.

LP77 Lee, D. T. ja Preparata, F. P., Location of a point in a planar subdivision and its applications. *SIAM Journal on Computing*, 6,3(1977), sivut 594–606.

NM95 Narkhede, A. ja Manocha, D., Fast polygon triangulation based on seidel's algorithm. Teoksessa *Grahpics Gems V*, Paeth, A. W., toimittaja, Morgan Kaufmann, 1995, sivut 394–397.

Sei91 Seidel, R., A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1,1(1991), sivut 51–64.

TVW88 Tarjan, R. E. ja Van Wyk, C. J., An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17,1(1988), sivut 143–178.