



TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma

OLLI ETUAHO
TYÖHISTORIAN TIETORAKENTEET JA SOVELLUSKOHTEET
BITTIKARTTAEDITORISSA
Kandidaatintyö

Tarkastaja: Mika Katara
Jätetty tarkastettavaksi 14.3.2009.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

ETUAHO, OLLI: Työhistorian tietorakenteet ja sovelluskohteet bittikarttaedito-
rissa

Kandidaatintyö, 26 sivua, 5 liitesivua

Maaliskuu 2009

Pääaine: Ohjelmistotekniikka

Tarkastaja: Mika Katara

Avainsanat: Bittikartta, työhistoria, tietorakenteet, sivellin, maalaus

Tämä työ käsittelee bittikarttapohjaisen piirto-ohjelman työhistorialta vaadittuja ominaisuuksia, joiden pohjalta arvioidaan eri vaihtoehtoja työhistorian tietorakenteiden ja toimintojen toteuttamiseksi. Lisäksi esitellään lyhyesti työhistoriaan liittyviä lisäsovellusmahdollisuuksia, kuten työskentelyprosessin toistamista animaationa, kuvan skaalausta ja useamman samanaikaisen käyttäjän editointia.

Kriittisimmiksi erotteleviksi ominaisuuksiksi työhistorian toteutusvaihtoehtojen välillä nousivat tuki valinnaiselle kumoamiselle ja toisaalta tuki operaatioille, jotka siirtävät pikseleiden väriarvoja toiselta kuvan alueelta toiselle. Erot käytännön toteutuksen yksinkertaisuudessa ovat myöskin hyvin merkittäviä, ja useamman peräkkäisen toiminnon kumoamisen nopeus vaihtelee eri toteutusvaihtoehtojen välillä. Erityisen olennaista on myös muistinkulutus, sillä bittikarttadata vie tyypillisesti runsaasti muistia. Muistinkulutukseen voidaan vaikuttaa toteutusvaihtoehdon valinnan lisäksi sen parametrien avulla erityisesti avainruutuihin perustuvan menetelmän tapauksessa.

Yleisenä tuloksena työhistorian toteutukseen tulisi käyttää mahdollisimman yksinkertaisia ratkaisuja, ja tavanomainen tilan tallentaminen on useimmiten riittävä menetelmä. Tilan tallentamiseen tarvittavaa rajatun pikselialueen tallentamista tutkittiin myös yksityiskohtaisemmin. Vertailluista menetelmistä parhaimpia tuloksia saavuttivat pikselien koordinaattien tallennus sekä pikselien jako dynaamisesti erikokoisiin ruutuihin.

Useamman samanaikaisen käyttäjän sovellusten kohdalla tilanne on hankalampi. Yksittäistä parasta toteutusvaihtoehtoa tässä tapauksessa ei ole helppo nimetä, vaan se on valittava sovellukselta haluttavien ominaisuuksien perusteella. Jatkotutkimusta vaaditaan, ennen kuin tässä työssä esitellyt menetelmät ovat sovellettavissa käytäntöön. Jatkotutkimusmahdollisuuksia tarjoaa myös laajempi tilastollinen tutkimus bittikarttaeditorien käyttötavoista, jonka avulla voitaisiin laskea eri algoritmien suoritusaike- ja muisti-vaatimuksia todellisella datalla.

ALKUSANAT

Erityiset kiitokset koepiirtäjinä toimineille Antti Jussilalle ja Jukka Länsinevalle. Lisäksi kiitokset työn kommentoinnista kandidaatintyöseminaariin osallistuneille, työn tarkastajalle Mika Kataralle sekä Tuure Laurinollille.

Olli Etuaho, 14.3.2009

SISÄLLYS

1. Johdanto.....	1
2. Työhistoria tietorakenteena.....	2
2.1. Operaatioiden luokittelu.....	2
2.2. Kumoamisoperaatioiden luokittelu.....	2
2.3. Työhistorian muistinhallinta.....	3
3. Bittikarttaeditorin perustoiminnot.....	4
3.1. Tietorakenteet.....	4
3.2. Toimintojen sivuvaikutukset ja yleiset ominaisuudet.....	5
3.3. Tasojen yhdistäminen.....	6
3.4. Sivellin.....	7
3.5. Toiminnon vaikutusalueen approksimointi.....	8
3.6. Vaikutusalueen approksimointimenetelmien kokeellinen arviointi.....	11
3.7. Suotimet.....	13
3.8. Kopiointi ja siirtäminen.....	14
4. Työhistorian lisäsovellukset bittikarttaeditorissa.....	15
4.1. Työhistorian toistaminen animaationa.....	15
4.2. Ylöspäin skaalaus työhistorian avulla.....	16
4.3. Muokkaus useammalla samanaikaisella käyttäjällä.....	17
5. Työhistorian toteutus bittikarttaeditorissa.....	18
5.1. Kuvan tilan tallentaminen.....	18
5.2. Toistaminen vektoriesityksen perusteella.....	19
5.3. Kumoaminen avainruutujen perusteella.....	20
5.4. Pällekkäisyysgraafi.....	20
6. Johtopäätökset.....	24
Lähteet.....	25
Liite 1: Java-algoritmi dynaamisen neliöihin jakamisen analysointiin.....	27
Liite 2: Analyysissä käytetyn bittikarttaeditorin käyttöliittymä.....	29
Liite 3: Koepiirtäjien piirrokset.....	30
Liite 4: Koepiirtäjien tehtävänanto.....	31

TERMIT JA MERKINTÄTAVAT

Antialiasointi	Grafiikan rasteroinnissa esiintyvien sahalaitaisten reunojen pehmennys.
Bittikartta	Pikseleistä koostettu kaksiulotteinen kuva.
Pikseli	Yksittäisen väriarvon omaava kuvapiste tietokoneen muistissa tai näytöllä.
Rasterointi	Kuvan muuntaminen vektorimuodosta tai analogisesta lähteestä bittikarttamuotoon.
Resoluutio	Bittikarttakuvan erottelutarkkuus, eli montako pikseliä mahtuu tietylle mittayksikölle.
Taso	Kuvan osa, joka sisältää yhden bittikartan sekä tiedot tämän piirtoasetuksista, kuten läpinäkyvyydestä tai efekteistä. Jotta kokonainen kuva saadaan aikaan, kuvan tasot on laskennallisesti yhdistettävä. Yksinkertainen vertauskuva on ajatella tasoa läpinäkyvänä kalvona ja koko kuvaa pinona näitä kalvoja.
Vektorigrafiikka	Matemaattisesti määritellyistä geometrisista muodoista koostettu grafiikka.
[a, b]	Suljettu lukuväli.
B[x, y]	Bittikartan indeksointi.
:=	Sijoitusoperaattori.
#010000FF	32-bittinen väriarvo heksamuodossa. Merkintä seuraa muotoa RGBA: ensimmäiset kaksi merkkiä esittävät punaista värikanavaa, toiset kaksi vihreää, kolmannet sinistä ja neljännet alfa-arvoa eli läpinäkyvyyttä.
round(x)	Pyöristys lähimpään kokonaislukuun.
Kertaluokkanotaatio	
O(f(n))	Suoritusajan tai muistinkulutuksen sanotaan olevan luokassa O(f(n)), mikäli löytyvät sellaiset n ₀ ja c, että kaikilla n > n ₀ tarkka suoritusajaksi tai muistinkulutus g(n) < c · f(n). O(f(n)) -merkintä antaa siis likimääräisen ylärajan suoritusajalle tai muistinkulutukselle suhteessa johonkin suureeseen n. Yleensä ollaan kiinnostuneita pienimmästä mahdollisesta ylärajasta.

Taulukoita indeksoidaan alkaen indeksistä 1, ellei toisin mainita. Algoritmeissa käytetään yksinkertaista pseudokoodia.

1. JOHDANTO

Nykyään suurin osa ammattimaisesta graafisesta työstä tehdään tietokoneella, ja usein jo alusta lähtien. Taiteilijan osaamisen lisäksi hyvät työkalut ovat tässä avainasemassa. Tietokonegrafiikka myös mahdollistaa paljon sellaista, mikä ei perinteisillä välineillä onnistu. Yksi piirtäjän tehokkaimmista työvälineistä on tämän työn toisessa luvussa esitelty työhistoria, joka tallentaa kuvalle tehdyt toiminnot ja mahdollistaa niiden kumoamisen ja toistamisen. Toimintojen kumoaminen ja toistaminen ovat nykyään vakio-ominaisuuksia lähes missä tahansa hyötyohjelmassa, mutta erilaisissa ohjelmissa on niille omanlaisiaan erityisvaatimuksia.

Graafiselle työlle on ominaista erityisesti se, että dataa on paljon. Raaka bittikarttadata vie yleensä ainakin kolme tai neljä tavua pikseliä kohden ja mahdollisesti enemmän. Toisaalta myös operaatiot saattavat olla laskennallisesti raskaita: kuva koostetaan useimmiten monista erillisistä bittikartoista eli tasoista, ja esimerkiksi siveltimen toiminnan mallintamiseen on kehitetty useita monimutkaisiakin menetelmiä. Bittikarttaeditorin perustietorakenteita ja työkaluja on käsitelty tarkemmin kolmannessa luvussa.

Toimintojen kumoamisen ja toistamisen lisäksi työhistoria mahdollistaa myös lisäominaisuuksia, joita ovat esimerkiksi usean samanaikaisen käyttäjän sovellukset tai työhistorian toistaminen jälkeinpäin animaationa, josta näkyy piirtäjän työskentelyprosessi. Näissä tapauksissa muistinkulutuksen merkitys korostuu entisestään, sillä tiedot joudutaan joko siirtämään verkon yli tai tallentamaan pysyvästi. Lisäominaisuuksia ja niiden yhteyksiä työhistorian rakenteisiin ja algoritmeihin on käsitelty luvussa neljä. Myös joitakin työhistorian toteutukseen vähemmän kiinteästi liittyviä lisäominaisuuksia esitellään lyhyesti.

Lopulta luvussa viisi tutkitaan sitä, millainen korkean tason tietorakenne olisi bittikarttaeditorissa sopiva ratkaisu työhistorian tallentamiseen, ja millaisia algoritmeja tämän yhteydessä tulisi käyttää, kun kaikki käytännön tekijät otetaan huomioon. Luvussa on esitelty neljä erilaista menetelmää työhistorian käsittelyyn lähtien yksinkertaisesta ja päätyen monimutkaisimpaan. Tietorakenteiden tarkastelussa otetaan huomioon tehokkuus ja muistinkulutus, mutta myös niiden asettamat rajoitteet ja vaatimukset työkalujen ja lisäominaisuuksien toteuttamiselle.

2. TYÖHISTORIA TIETORAKENTEENA

Työhistoria on yhden käyttäjän sovelluksessa tyypillisesti lineaarinen lista-tyyppinen tietorakenne, jossa perusyksikkönä on suoritettu toiminto. Toimintoja on listassa kahdenlaisia: suoritettuja toimintoja sekä kumottuja toimintoja. Toimintojen erittely näihin ryhmiin onnistuu helpoiten jakamalla lista kahdeksi listaksi, mutta joustavampi vaihtoehto on säilyttää lineaarinen rakenne ja merkitä ryhmä jokaiseen toimintoon erikseen.

Tässä luvussa käydään läpi työhistoriaan yleisesti liittyviä käsitteitä. Ensin esitellään eri operaatiotyypit, ja erityisesti kumoamisoperaatiot jaetaan kolmeen erilaiseen luokkaan Sunin (2000) esittämään tapaan. Lisäksi esitellään lyhyesti työhistoriaan liittyvä muistinhallinta ja tähän liittyvä leikkausoperaatio.

2.1. Operaatioiden luokittelu

Käyttäjän perusoperaatioita on kolme:

- Uusien toimintojen suorittaminen, jolloin toiminto lisätään suoritettuihin toimintoihin.
- Toimintojen kumoaminen, jolloin toiminto siirtyy suoritetuista toiminnoista kumottuihin.
- Kumottujen toimintojen toistaminen, jolloin toiminto siirtyy kumotuista toiminnoista suoritettuihin.

Uusien toimintojen suorittaminen on työhistorian kannalta tietyllä tapaa sama asia kuin toiminnon toistaminen, sillä molemmissa tapauksissa pätevät samat jälkiehdot: toiminnon tulee löytyä työhistoriasta suoritettujen toimintojen puolelta, ja historiassa tulisi olla tieto, miten toiminto voidaan kumota ja kumoamisen jälkeen toistaa. Kumottujen toimintojen puolelta kumoamiseen liittyvät tiedot voidaan muistin säästämiseksi poistaa ja luoda sitten uudelleen, jos toiminto toistetaan.

2.2. Kumoamisoperaatioiden luokittelu

Edellä esitetyn jaon lisäksi Sun (2000) jakaa yhteistyöohjelmia käsittelevässä tutkimuksessaan kumoamisoperaatiot kolmeen luokkaan: *yhden askeleen* kumoamiseen, *kronologiseen* kumoamiseen ja *valinnaiseen* kumoamiseen.

Yhden askeleen kumoaminen on näistä tavallisin. Siinä kumotaan suoritetuista toiminnoista viimeisin, ja samalla siirretään tämä kumottuihin toimintoihin mahdollista toistoa varten. Kronologisessa kumoamisessa, joka löytyy myös joistakin sovellusohjel-

mista, kumotaan useampia peräkkäisiä toimintoja kerralla. Yksinkertaisimmillaan kronologinen kumoaminen voidaan toteuttaa useampana peräkkäisenä yhden askeleen kumoamisena, mutta joskus kronologinen kumoaminen on mahdollista toteuttaa tätä tehokkaammin.

Erikoisempi luokka on valinnainen kumoaminen, jossa mikä tahansa historiassa suoritettuna oleva toiminto voidaan kumota kumoamatta välttämättä tämän jälkeen suoritettuja toimintoja (Sun 2000). Valinnaisen kumoamisen toteuttaminen tehokkaasti saattaa olla huomattavasti hankalampaa kuin toimintojen kumoamisen järjestyksessä. Bittikarttapohjaisille sovelluksille tätä ovat tutkineet muiden muassa Wang et al. (2002). Erityisesti useamman käyttäjän sovelluksissa valinnainen kumoaminen vaikuttaa merkittävästi sovelluksen käyttökokemukseen, kuten näytetään myöhemmin.

2.3. Työhistorian muistinhallinta

Riippumatta työhistorian toteutuksesta ja muista ominaisuuksista historiatiedot vievät yleisessä tapauksessa tallennettujen toimintojen määrään verrannollisen määrän muistia. Tästä syystä historian kasvaessa on valittava, kuinka suuri osa tiedoista lopulta säilytetään muistissa.

Yleisesti ottaen käyttäjällä on taipumusta kumota vain viimeisimpiä toimintoja, joten useissa sovellusohjelmissa toimintojen kumoamiseen liittyviä tietoja säilytetään muistissa vain rajallinen määrä. Historian toimintoja siis poistetaan vanhempien toimintojen päästä sitä mukaa, kun uusia lisätään. Jatkossa tästä poisto-operaatiosta käytetään nimeä työhistorian leikkaaminen. Leikkauskriteerinä voi olla esimerkiksi historiassa olevien toimintojen määrä tai niiden yhteensä kuluttama muisti. Tulee kuitenkin huomata, että vaikka toimintoja leikattaisiin historiasta, toistomahdollisuus saattaa viedä nopeaan kumoamiseen liittyviä tietoja vähemmän tilaa, ja toistomahdollisuuden säilyttäminen koko työhistorian ajalta on joidenkin erikoissovellusten kannalta mielekästä. Tästä syystä leikkaaminen voidaan määritellä koskemaan ainoastaan toimintojen kumoamiseen liittyviä tietoja, ja käsitellä toistamiseen liittyvä data erikseen.

Kumottujen toimintojen puolelta säilytetään kaikki toiminnot niin kauan, kunnes niiden toistaminen ei enää ole käytettävyyden kannalta järkevää. Usein ainakin käyttäjän suorittaessa uusia toimintoja kaikki kronologisesti kumotut toiminnot voidaan tyhjentää historiasta. Valinnaisesti kumotut toiminnot taas tulisi säilyttää historiassa vähintäänkin siihen asti, kunnes ne poistuvat historian vanhimpien toimintojen joukosta normaalien leikkauskriteerien mukaan.

3. BITTIKARTTAEDITORIN PERUSTOIMINNOT

Bittikarttaeditoriksi käsitetään tässä työssä sellainen grafiikkaeditori, jossa olennaisin osa käsiteltävästä datasta on tallennettu bittikarttoina. Bittikarttoja täydentämässä voi olla myös jonkin verran muita parametreja, mutta nämä vaikuttavat koko kuva-alaan kuvan yksittäisten muotojen sijaan. Esimerkkejä tällaisista editoreista ovat Photoshop (Adobe Systems 2008) ja avoimen lähdekoodin GIMP (The GIMP Team 2009). Photoshopissa on myös jonkin verran vektorigrafiikkaominaisuuksia, mutta ohjelman pääpaino on kuitenkin bittikarttagrafiikassa. Lisäksi on olemassa lukuisia muita vastaavaan tarkoitukseen kehitettyjä ohjelmia, jotka eroavat usein yksityiskohdiltaan ja erikoistumis-kohteeltaan mutta harvemmin peruseriaatteiltaan.

Seuraavassa kohdassa esitellään yleisluontoinen, joskin hieman yksinkertaistettu malli nykyaikaisen bittikarttaeditorin olennaisista tietorakenteista. Tätä mallia käytetään jatkossa pohjana algoritmien toteutuskelpoisuuden, muistinkulutuksen ja laskennallisen vaativuuden arvioinnille. Tämän jälkeen käsitellään bittikarttaeditorin toiminnoille yhteisiä piirteitä, joista edetään tasojen yhdistämisoperaation kautta yksittäisten toimintotyyppien ominaisuuksien käsittelyyn.

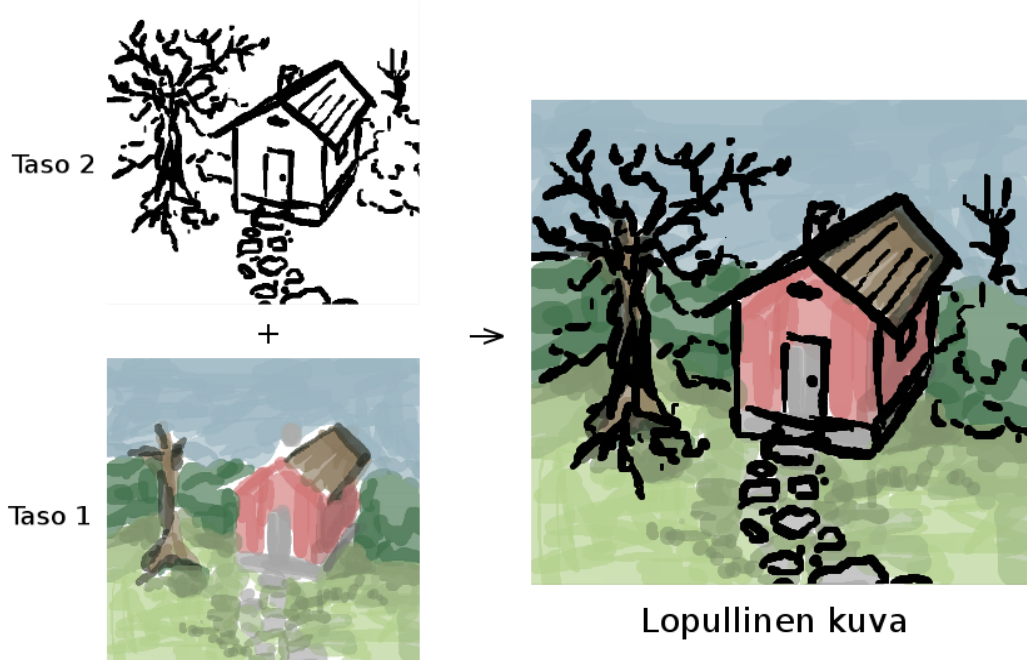
3.1. Tietorakenteet

Bittikarttadata oletetaan tässä työssä yksinkertaisuuden vuoksi aina 32-bittiseksi ja sisältämään neljä 8-bittistä värikanavaa: punaisen, sinisen, vihreän ja alfakanavan, joka sisältää pikseleiden läpinäkyvyystiedon. Tulokset ovat kuitenkin yleistettävissä indeksoiduille väreille, joka on erikoistapaus tästä, tai useampibittisille värikanaville. Kukin 8-bittinen värikanava määrittelee väriarvon lineaarisesti kokonaislukuna väliltä $[0, 2^8-1]$.

Nykyaikaisessa bittikarttaeditorissa kuva on koostettu useista päällekkäisistä bittikartoista eli tasoista, jotka yhdistetään laskennallisesti lopullisen kuvan aikaansaamiseksi. Bittikarttaeditorien kehitys on johtanut siihen, että jo pelkät tasoihin liittyvät ominaisuudet saattavat olla melko monimutkaisia. Esimerkiksi Photoshopissa tasoja voi ryhmitellä ja liittää toistensa alitasoiksi, jolloin tason läpinäkyvyys vaikuttaa alitason läpinäkyvyyteen. Lisäksi tasoihin on mahdollista liittää erilaisia erikoisefektejä. Tässä työssä käsitellään vain yksinkertaista lineaarista tasorakennetta, ja monimutkaisempien tasorakenteiden käsittely on rajattu työn ulkopuolelle.

Tämän työn puitteissa myös oletetaan kuvasta joitakin yleisesti käytössä olevia peruseriaatteita. Kaikille tasoille oletetaan sama resoluutio, ja niiden muodoksi suorakulmio. Tasot saattavat kuitenkin olla lopullista kuva-ala suurempia ja sijaita osittain tai kokonaan sen ulkopuolella. Koordinaattiavaruuden koon ylärajaksi otetaan $2^{16} \cdot 2^{16}$

pikseliä. Näin pikselin kokonaislukuarvoisten x- ja y-koordinaattien tallentamiseen riittää 32 bittiä eli 4 tavua muistia. Esimerkki tasoista koostetusta kuvasta on kuvassa 3.1.



Kuva 3.1: Yksinkertainen esimerkki tasoista, kuva "Talo".

Bittikarttadatan lisäksi tasoille määritellään kaksi ominaisuutta: intensiteetti-arvo I väliltä $[0,1]$, sekä mielivaltainen värinsekoitusfunktio $f(c_1, c_2, I_2) \rightarrow c$, jossa c -arvot ovat kelvollisia pikselien väriarvoja ja I -arvo tason intensiteetti-arvo. Normaalisissa käytössä värinsekoitusfunktio valitaan kokoelmasta ennalta toteutettuja vaihtoehtoja.

Tasojen lisäksi olennaisia tietorakenteita ovat työkalujen asetukset. Kaikille työkaluille yhteisiksi asetuksiksi määritellään yleiseen tapaan kaksi väriarvoa: edustaväri ja taustaväri. Loput asetukset ovat työkalukohtaisia, ja saattavat vaihdella työkalujen välillä suurestikin. Näistä oletetaan ainoastaan, että asetusten vaatima tila on merkittävästi pienemmässä suuruusluokassa kuin bittikarttadatan vaatima tila, joten työkaluasetusten tallennuksen muistinkulutus voidaan jättää huomiotta.

3.2. Toimintojen sivuvaikutukset ja yleiset ominaisuudet

Bittikarttaeditorin laskennallisista operaatioista osa on käyttäjän itsensä suorittamia toimintoja, ja osa näiden sivutuotteena suoritettavia. Jokaisen muutosoperaation yhteydessä tulee ainakin päivittää lopullista, ruudulla näkyvää kuvaa, mitä on käsitelty tarkemmin seuraavassa kohdassa. Lisäksi sivutuotteena suoritetaan myös työhistoriaan liittyvät edellisessä luvussa määritellyt operaatiot, joiden toteutusta on käsitelty syvällisemmin myöhemmin.

Lähes kaikille toiminnoille on yhteistä se, että ne hävittävät kuvasta informaatiota, eli yksinkertaisimmillaan korvaavat pikselien väriarvoja toisilla. Tämä on olennaisia toimintojen kumoamisen yhteydessä – jos tiedetään toiminnon lopputulos sekä mikä toi-

minto on tehty, pelkästään tästä tiedosta ei voida yleisessä tapauksessa päätellä kuvan tilaa ennen toimintoa. Toisin sanottuna toiminnoille ei voida määrittää käänteistoimintoa. Tilanne on erilainen kuin vektoripohjaisissa piirto-ohjelmissa, joissa toiminnot ovat yleensä kumottavissa yksinkertaisen käänteistoiminnon avulla (Wang et al. 2002).

Otetaan ei-triviaalina esimerkkinä pikselin värin sekoittaminen mustaan suhteessa 50% - 50%. Oletetaan, että lopputuloksena saatavan pikselin väriarvo on #010000FF. Mikäli toiminto yritettäisiin kumota tämän tiedon perusteella, aiempi punaisen värikanavan arvo (x) pitäisi päätellä yhtälöstä $1 = \text{round}(0,5 \cdot 0 + 0,5 \cdot x)$. Mikäli pyöristystä ei olisi, yhtälöstä saataisiin yksinkertaisella algebralla ratkaisuksi $x = 2$. Nyt kuitenkin sekä $x = 1$ että $x = 2$ ratkaisevat yhtälön, joten pikselin punaisen värikanavan aiempi tila voidaan päätellä vain likimääräisesti. Vaikka pohjalla ollut väri vaikutti lopputuloksena syntyvään väriin, tässäkään tapauksessa toiminnon kumoaminen ei onnistu ilman tarkkaa tietoa pikselin tilasta ennen toimintoa.

3.3. Tasojen yhdistäminen

Käyttäjän ruudulla näkemä lopullinen kuva on aina yhdistelmä kaikista näkyviksi asetetuista tasoista (ks. kuva 3.1). Koska tasojen yhdistäminen koko kuvan alalta on melko raskas operaatio, yleisesti ottaen yhdistetty kuva kannattaa tallentaa erikseen muistiin ja päivittää siitä vain ne osat, jotka on tarpeen päivittää. Joka tapauksessa tasojen yhdistämisen suorituskyky on olennaista käyttäjän saaman palautteen kannalta. Lisäksi käyttäjä saattaa haluta yhdistää joukon tasoja manuaalisesti yhdeksi tasoksi työskentelyn nopeuttamiseksi tai helpottamiseksi.

Yhdistämisoperaatio suoritetaan halutulle pikselijoukolle pikseli kerrallaan. Määritellään n_T tasojen määräksi, B_i tason i bittikartaksi, f_i tason i värinsekoitusfunktioiksi sekä I_i tason i intensiteetiksi. Lopullinen kuva luodaan bittikarttaan R , ja lisäksi kuvalle on määritelty erityinen pohjaväri c_0 , jonka päälle kaikki tasot yhdistetään. Nyt yhdistämisoperaatio yhdelle pikselille koordinaateissa x, y onnistuu seuraavalla algoritmilla (kuva 3.2). Tästä yksinkertaisesta algoritmista nähdään, että yhdistämisoperaation kompleksisuus pikseliä kohden on luokassa $O(n_T)$, ja siten kokonaisuudessaan $O(n_T n_P)$, missä n_P on pikseleiden määrä. Muistin suhteen kirjoitusoperaatioita R :ään tehdään $(n_T + 1)n_P$, joista n_P on alustuksia värillä c_0 . Lisäksi tehdään $n_T n_P$ lukuoperaatiota R :lle sekä n_P lukuoperaatiota kullekin B_i . Funktio f_i oletetaan vakioaikaiseksi.

```
foreach (x, y) in yhdistettävät_pikselit:
    R[x, y] := c0
    i := 1
    while (i < nT)
        R[x, y] := fi(R[x, y], Bi[x, y], Ii)
        i := i + 1
    end while
end foreach
```

Kuva 3.2: Tasojen yhdistäminen.

Algoritmi kuvassa 3.2 käy läpi kaikki kuvassa olevat tasot ja yhdistää kunkin tason pikselit pohjalla oleviin pikseleihin käyttäen tasolle määriteltyä värinsekoitusfunktiota. Tämä algoritmi toteuttaa tasojen yhdistämisen siten kuin se toimii lähes kaikissa bittikarttaeditoreissa, joskin toteutuksen yksityiskohdat ja joustavuus saattavat vaihdella. Todellisissa sovelluksissa kuva saatetaan jakaa sisäisesti osiin ja kuvaformaatti saattaa vaihdella jopa yksittäisten tasojen kesken, mikä monimutkaistaa algoritmia suurestikin.

Lisäksi todellisessa sovelluksessa uloin iterointi saatetaan suorittaa tasoille ja sisempi iterointi pikseleille tässä esitetyn järjestyksen sijaan. Tällä pystytään optimoimaan muistin käyttöä, joka tyypillisessä tapauksessa nopeutuu luettaessa tai kirjoitettaessa peräkkäisiä muistialkioita. Tässä yhteydessä tulee huomata, että tämänkaltaisen tasojen yhdistäminen on lähes täydellisesti rinnakkaistettavissa. Vapaasti saatavilla oleva reaali maailman esimerkki löytyy GIMP:n funktiosta `gimp-image-merge-layers` (Kimball & Mattis 1995).

3.4. Sivellin

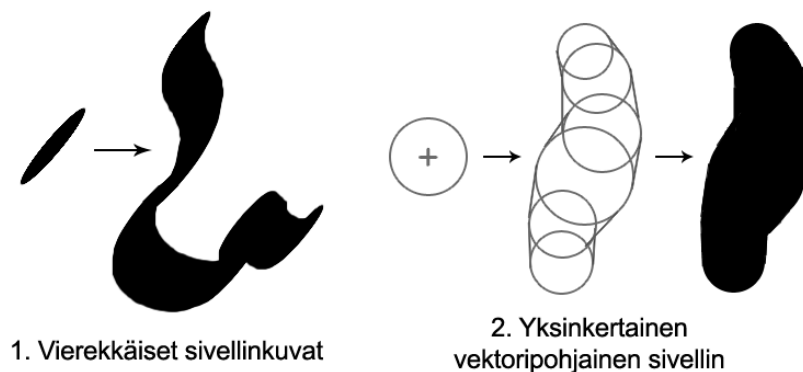
Siveltimellä tarkoitetaan tässä yhteydessä mitä tahansa sellaista työkalua, joka tuottaa kuvaan cursorin liikkeitä seuraavia käyriä. Sivellin on mahdollisesti kaikkein keskeisin bittikarttaeditorin työkaluista, ja siten myös tietorakenteet tulisi toteuttaa sellaisiksi, että ne tukisivat siveltimen toimintaa mahdollisimman hyvin. Erityisen olennaista on, että käyttäjä saa sivellintä käyttäessään välittömän palautteen, ja myös työhistoriaan liittyvien operaatioiden tulisi olla mahdollisimman nopeita. Sivellinoperaatioita myös suoritetaan yksittäiselle kuvalle mahdollisesti hyvinkin suurina määrinä, joten operaation työhistoriatietojen muistinkulutus ansaitsee erityistä huomiota.

Erilaisia sivellintyökalun toteuttamismahdollisuuksia on tutkittu laajalti, eikä vaihtoehtoista voida nimetä mitään yksittäistä tapausta, jonka perusteella työkalun suoritusaikaa voitaisiin edes esimerkin kaltaisesti analysoida. Joitakin yhteisiä piirteitä sivellintoteutuksilla kuitenkin on. Suurimmalla osalla on joitakin yhteisiä parametreja, kuten intensiteetti, koko sekä siveltimen muoto. Merkittävä osa moderneista toteutuksista on myös kehitetty paineentunnistavaa syötelaitetta silmällä pitäen, eli ne sallivat x- ja y-koordinaattien lisäksi myös muita piirtämisen aikana muuttuvia dynaamisia parametreja. Nykyään tyypillinen esimerkki tällaisesta syötelaitteesta on paineentunnistava piirtopöytä, joka koostuu tasaisesta sensoripinnasta sekä pinnalla käytettävästä erityisestä kynästä.

Eräs yksinkertaisimmista perussiveltimen malleista on cursorin piirtämää käyrää seuraava jono vierekkäisiä sivellinkuvia. Tämän sovellusmahdollisuuksia on tutkinut laajemmin muun muassa Whitted jo vuonna 1983 julkaistussa tutkimuksessaan. Ratkaisu on sikäli hyvä, että sillä saavutetaan suuri joustavuus tinkimättä kuitenkaan suorituskyvystä. Sivellinkuva voi olla mikä kuva tahansa, ja käytettäessä suuriresoluutioista sivellinkuvaa voidaan simuloida myös antialiasointia, eli rasteroinnissa esiintyvän sahalaitaisuuden poistoa. Ratkaisu tosin tarjoaa pelkän approksimaation verrattuna tarkkoihin vektoripohjaisiin muotoihin perustuvaan antialiasointiin (Whitted 1983, s. 153).

Lisäksi bittikartan avulla määritellyn siveltimen kokoa voi skaalata vain tiettyyn määrättyyn ylärajaan asti, kunnes epätarkkuus alkaa näkyä. Mikäli pohjalla oleva bittikartta kuitenkin voidaan generoida tarvittaessa suurempaan kokoon vektoriesityksen perusteella, tämä ongelma poistuu.

Mikäli halutaan määrittellä siveltimen jälki tarkasti vektoreiden perusteella, toimiva malli on piirtää käyrän varrelta valittaviin näytepisteisiin ympyrä ja yhdistää nämä ympyrät reunojen tangentteja sivuavilla puolisuunnikkailla. Tämä lähestymistapa ei tarjoa läheskään edelliseen vaihtoehtoon verrattavissa olevaa joustavuutta, mutta sen avulla voidaan piirtää tarkasti minkä tahansa kokoinen kiinteä siveltimenveto. Antialiasointiin voidaan käyttää mitä tahansa suorille viivoille ja ympyröille tunnettua menetelmää, kuten esimerkiksi Wun (1991) kehittämää nopeaa menetelmää.



Kuva 3.3: Kaksi yksinkertaista sivellinmallia.

Esimerkkejä sivellinmalleista on esitetty kuvassa 3.3. Vasemmalla vinon ellipsin kuvaa on kopioitu vierekkäin, jolloin tuloksena on kalligrafiatyypinen siveltimenveto. Oikealla on esitys vektorimuotoisesta siveltimestä, jossa näytepisteisiin piirrettyjen ympyröiden välit on täytetty tangentteja sivuavien puolisuunnikkaiden avulla.

Monimutkaisempiakin sivellintoteutuksia on tutkittu. Hsu & Lee (1994) ehdottavat Skeletal strokes -sivellintä, joka on alkujaan suunnattu vektoripohjaiseen ympäristöön, mutta olisi sovellettavissa myös bittikarttaeditoriin. Tässä järjestelmässä sovitaan mielivaltaisen kuva siveltimen tuottamalle käyrälle. Kuva voidaan määrittellä myös rekursiivisesti, jolloin tuloksena on fraktaalinen sivellin.

3.5. Toiminnon vaikutusalueen approksimointi

Työhistorian kannalta toimintojen tarkkaa toteutusta olennaisempaa on, kuinka kuva muuttuu toiminnon käytön seurauksena. Määrittellään toiminnolle tätä mittaamaan käsite vaikutusalue, joka on niiden pikseleiden tai pikselikoordinaattien joukko, joiden väriarvoon toiminnolla on vaikutusta.

Kattavaa tilastollista tutkimusta aiheesta ei ole saatavilla, mutta yleisenä suunta-
viivana voidaan sanoa, että esimerkiksi yksittäisen siveltimenvetäön vaikutusalue on ainakin koko kuva-alaa merkittävästi pienempi. Kun toiminnon vaikutusalueita halutaan mallintaa työhistoriaa varten, mahdollisuuksia on kaksi: joko käsitellään vaikutusalueen

pikselit tarkasti, tai sitten etsitään näille jokin yläraja. Määritellään ylärajalle käsite täyttösuhde, joka kertoo, kuinka tiukka yläraja on. Mitä korkeampi täyttösuhde on, sitä vähemmän tallennettua dataa menee hukkaan, ja siten myös vähemmän pikseleitä tarvitsee käsitellä turhaan, kun vaikutusalueen datalle täytyy tehdä laskutoimituksia.

$$\text{täyttösuhde} = \text{vaikutusalueen } n_p / \text{ylärajan } n_p$$

Lisäksi voidaan arvioida ylärajan suhteellista tilankulutusta ideaaliratkaisun suhteen: jokaisesta pikselistä on tallennettava ainakin väriarvo, joten ideaalisena ratkaisuna voidaan pitää sellaista, joka ei kuluta tämän lisäksi lainkaan tilaa. Määritellään ideaaliratkaisun tilankulutus tavuina S_I . Lisäksi määritellään varsinaisen tilankulutuksen (S) funktiona suhteellinen tilankulutus RS :

$$S_I = n_p \cdot 4$$

$$RS = S / S_I$$

Helpoiten käsiteltävä yläraja on etsiä kuvasta pienin mahdollinen akseleiden suuntainen suorakulmio, johon vaikutusalueen pikselit mahtuvat. Paremman täyttösuhteen vaihtoehtona suorakulmaiselle akseleiden suuntaiselle ylärajalle olisi vapaasti käännettävissä oleva suorakulmainen yläraja, mutta tällaisen generointi on laskennallisesti monimutkaisempi operaatio, ja esimerkiksi voimakkaasti kaareutuvat siveltimenvedot saattavat silti synnyttää huonon täyttösuhteen. Tästä syystä vapaan kulman suorakulmainen yläraja sivuutetaan. Akseleiden suuntaisen suorakulmion suurimpana etuna on sen algoritmisen yksinkertaisuus, joka koskee sekä rakenteen luontia että mahdollisia lisäoperaatioita, kuten leikkauksen laskemista toisen vastaavan ylärajan kanssa. Rakenteen kuluttaa tilaa 8 tavua koordinaateille sekä ulottuvuuksille ja 4 tavua tallennettua pikseliä kohden, siis yhteensä:

$$S = 8 + \text{leveys} \cdot \text{korkeus} \cdot 4$$

Toisena vaihtoehtona on jakaa suorakulmainen vaikutusalueen yläraja pienempiin $h \cdot h$ pikselin neliöihin, joista otetaan vaikutusalueen ylärajaan mukaan ne, joille toiminnon vaikutusalue ulottuu. Tämä vaihtoehto antaa tasaisen hyvän täyttösuhteen kaikentyyppisille operaatioille varsinkin pienellä neliökoolla, mutta toisaalta jokaiseen neliöön liitettävät ylänurkan koordinaatit lisäävät tilavaatimuksia neliötä kohden 4 tavulla. Rakenteen etuna on edelleen yksinkertaisuus, ja neliöt voidaan kohdistaa myös koko kuvalle laskettuun ruudukkoon, jolloin leikkauksen laskeminen toisen neliöjoukon kanssa on yksinkertaista sekä tehokasta. Algoritmeihin ei tässä puututa tarkemmin. Samankokoisiin neliöihin jaetun alueen tilankulutus tavuina on:

$$S = n_{\text{neliöt}} \cdot 4 + n_{\text{neliöt}} \cdot h^2 \cdot 4$$

Variaationa edellisestä voidaan käyttää sekaisin erikokoisia neliöitä esimerkiksi rekursiivisen rakenteen avulla generoituina. Tällöin jokaisen neliön lisätilavaatimuksiksi tulevat neliön koko, jolle riittää 1 tavu, ja neliön koordinaatit, joille tulee varata 4 tavua. Erikokoisiin neliöihin jaetun alueen haittapuolena on neliöiden generoinnin monimutkaisuus, ja myöskin lisäoperaatioiden monimutkaisuus. Tämän työn analyysi-

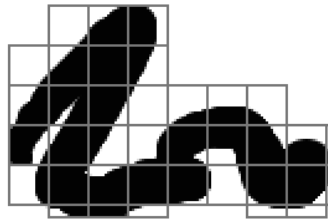
sissä käytettiin generointiin rekursiivista neljään neliöön jakoa, johon käytetty Java-algoritmi löytyy liitteestä 1. Algoritmi perustuu Quadtree-tietorakenteen ajatukseen (Weisstein). Erikokoisiin neliöihin jaetun alueen tilankulutus tavuina on:

$$S = n_{neliöt} \cdot 5 + \sum h_i^2 \cdot 4$$

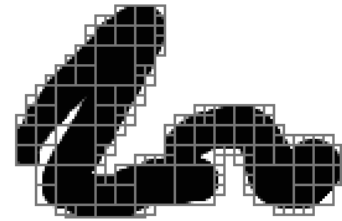
Neliöihin jakoa voidaan vielä optimoida hieman tallentamalla neliöistä vain ne pikselit, jotka jäävät suuremman suorakulmaisen vaikutusalueen sisäpuolelle. Tämä vaatii lisäparametreina ainoastaan suorakulmaisen vaikutusalueen koordinaatit ja ulottuvuudet (8 tavua), sillä vajaan kokoisten neliöiden koko voidaan päätellä suuremman suorakulmaisen vaikutusalueen koosta.



1. Suorakulmainen yläraja



2. 16x16 pikselin neliöt



3. Dynaamiset neliöt

Kuva 3.4: Vaikutusalueen erilaisia ylärajoja annetulle siveltimenvedolle.

Eri ylärajoja on havainnollistettu kuvassa 3.4. Kuvassa on suorakulmainen yläraja (1), jako suorakulmaisen ylärajan sisällä 16x16 pikselin neliöihin (2) sekä dynaaminen neliöihin jako (3). Kuvan dynaaminen jako ei täysin vastaa analyysissä käytetyn algoritmin tuottamaa tulosta, mutta on kuitenkin suuntaa-antava.

Jos otetaan mukaan täsmälleen vaikutusalueella olevat pikselit, vaihtoehtona on tallentaa pikseleiden indeksit sellaisenaan, mutta tällainen rakenne tuo mukanaan merkittävän määrän omia tilavaatimuksiaan. Mikäli jaetaan vaikutusalue $2^8 \cdot 2^8$ pikselin neliöihin, joiden sisällä pikselikoordinaatit tallennetaan, jokaisen pikselin koordinaatit vaativat 16 bittiä tilaa pikselin 32-bittisen väridatan lisäksi. Jokaisen $2^8 \cdot 2^8$ alueen koordinaattien tallennukseen pitää lisäksi varata 32 bittiä. Pikseleiden koordinaattien tallentamisen tilankulutus tavuina on:

$$S = n_p \cdot 6 + n_{neliöt} \cdot 4$$

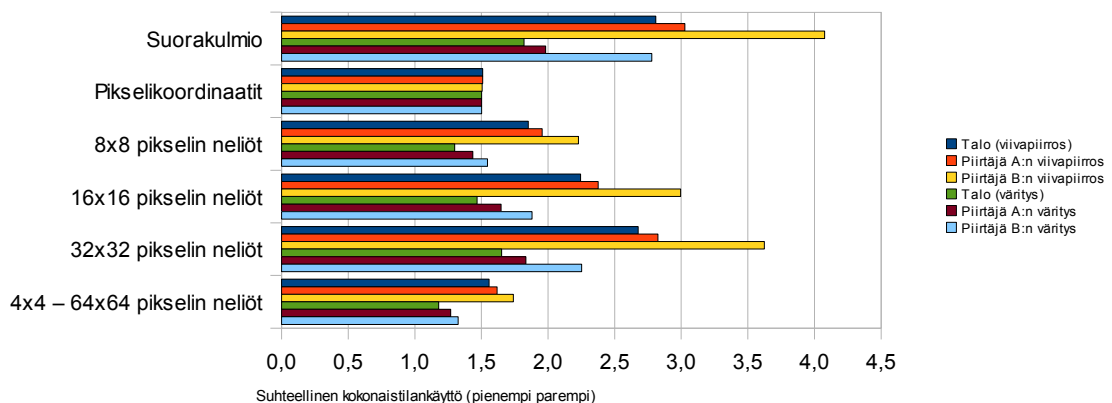
Mikäli halutaan todella minimoida muistinkulutus, voidaan pikselikoordinaattien tallentamisen sijaan kehittää menetelmä, joka liittää toiminnon käsittelemiin pikselikoordinaatteihin indeksit syöteparametrien perusteella. Tällöin operaatiossa katoava pikselidata voidaan tallentaa ideaalisella tilankulutuksella yksiulotteisena taulukkona, ja taulukon alkioihin liittyvät pikselikoordinaatit voidaan laskea kumoamisen yhteydessä uudestaan syöteparametrien perusteella, jotka on myös tallennettava. Lisätilavaatimukset riippuvat tällöin ainoastaan syötedatan monimutkaisuudesta. Kadonneen informaation käsittelyn raskaus tulee kuitenkin tällöin vahvasti riippuvaiseksi toiminnon itsensä suoritusajasta.

3.6. Vaikutusalueen approksimointimenetelmien kokeellinen arviointi

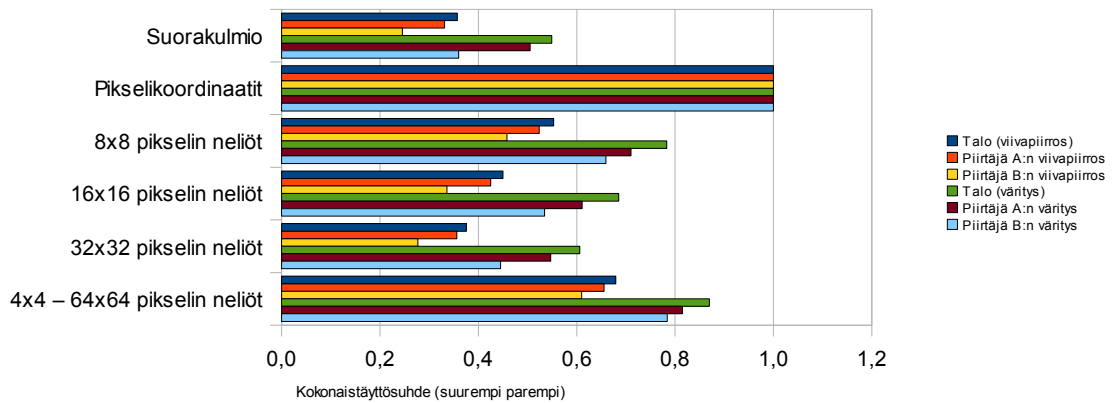
Täyttösuhteen ja tilankulutuksen vaihtelua eri vaikutusalueen ylärajoilla arvioitiin kokeellisesti Javalla kirjoitetulla bittikarttaeditorilla, jossa käyttäjälle annettiin kaksi 400 · 400 pikselin tasoa sekä maksimissaan 32 pikselin levyinen sivellin. Sivellin perustui vierekkäisiin sivellinkuviin ja sen koko määräytyi syötelaitteen paineen perusteella. Editorin käyttöliittymäkuva löytyy liitteestä 2.

Ensinnäkin tekniikoita arvioitiin piirtämällä yksinkertainen piirros talosta ja puutarhasta (kuva 3.1), ensin viivapiirroksena ja sitten värittämällä. Kuvaan kului 246 siveltimevetoa, joille laskettiin kokonaistäyttösuhte annettulle tekniikalle ja väriarvojen sekä koordinaattien tarkka suhteellinen kokonaistilankulutus. Täyttösuhte laskettiin jakamalla vaikutusalueiden pikseleiden kokonaismäärä koko työhistoriaan tallennettujen pikseleiden määrällä. Suhteellinen kokonaistilankulutus laskettiin jakamalla kunkin menetelmän summattu tilankulutus koko summattulla ideaalitulankulutuksella. Lisäksi laskettiin yksittäisten toimintojen täyttösuhteille tilastollinen varianssi.

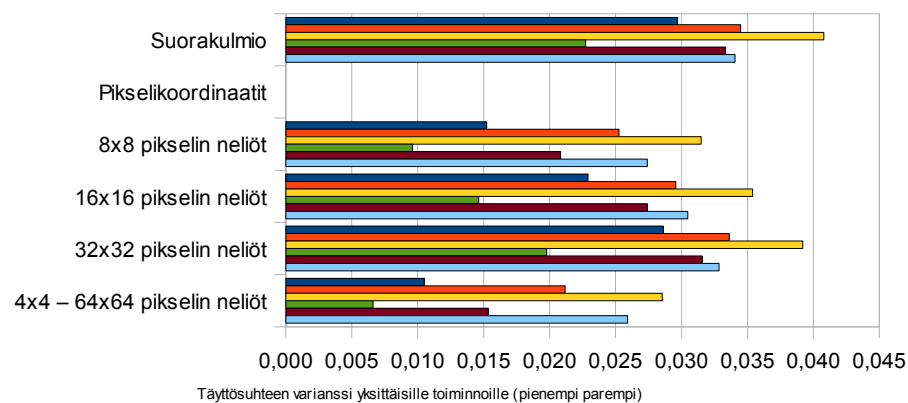
Talo-kuvan lisäksi lähdemateriaalissa oli neljä piirrosta kahdelta ulkopuoliselta koehenkilöltä. Koepiirtäjä A oli 3D-mallinnuksen medianomi-opiskelija, ja koepiirtäjä B harrastaja. Koehenkilöt käyttivät kuvien piirtämiseen omaa laitteistoaan Windows-ympäristössä. Näin päästiin arvioimaan käyttäjän tekniikan sekä käyttöympäristön vaikutusta tuloksiin. Syntyneet kuvat sekä koehenkilöille annettu tehtävänanto löytyvät liitteistä 3 sekä 4 ja tulokset kuvista 3.5, 3.6 sekä 3.7. Kokonaisuudessaan otos on varsin pieni, vain kuusi erillistä kuvaa, mutta erot eri käyttäjien välillä näkyvät silti. Tulokset ovat saman piirtäjän kohdalla samankaltaisia sekä viivapiirrokselle että väritykselle, joten työskentelytapa selkeästikin vaikuttaa muistinkulutukseen.



Kuva 3.5: Suhteellinen kokonaistilankulutus eri vaikutusalueen ylärajoille kuuden kuvan otoksella.



Kuva 3.6: Kokonaistäyttösuhde eri vaikutusalueen ylärajoille kuuden kuvan otoksella.



Kuva 3.7: Täyttösuhteen tilastollinen varianssi yksittäisille toiminnoille eri vaikutusalueen ylärajoille kuuden kuvan otoksella.

Kuvasta 3.7 on havaittavissa, että viivapiirroksen täyttösuhdeiden vaihteluväli on tyypillisesti suurempi kuin maalauksenomaisen värityksen. Lisäksi viivapiirroksen suhteellinen tilankulutus on säännönmukaisesti suurempi (kuva 3.5) ja kokonaistäyttösuhde pienempi (kuva 3.6).

Suorakulmion muotoisen alueen tilankulutus oli säännönmukaisesti suurempi kuin jaettaessa alue edelleen ruutuihin (kuva 3.5). Parhaan kokonaistuloksen tilankulutuksen suhteen tarjosivat dynaamisesti ruudukkokokoa säätelevä rakenne sekä hieman yllättäen pikselikoordinaattien tallennus. Dynaaminen ruudukkokoko pärjäsi paremmin värityksen kanssa, kun taas koordinaattien tallennus voitti vertailun viivapiirrosten kohdalla johtuen ilmeisestikin viivojen kapeudesta. Pikselikoordinaattien tallennus saavutti luonnollisesti myös ideaalisen täyttösuhteen ja siten nollavarianssin.

Tuloksiin tulee suhtautua jossain määrin kriittisesti, sillä siveltimen säädettävyyden oli puutteellista eikä muita työkaluja ollut kumoamisen ja toistamisen lisäksi. Käyttäjän työskentelytapa analyysissä käytetyllä editorilla ei siten välttämättä vastaa sitä, miten käyttäjä työskentelee kehittyneemmällä työkaluilla. Myös käyttöön annettu kuva-ala oli suhteellisen pieni, mikä suosii pienikokoisille siveltimenvedoille soveltuvia menetelmiä. Myöskin laskettu varianssi koskee kaikenkokoisia siveltimenvedoja, vaikka merkityksellisempi tulos olisi verrata varianssia samankokoisilla, mutta erimuotoisilla siveltimenvedoilla. Lopulta suositeltavin menetelmä on jakaa vaikutusalue neliöihin.

3.7. Suotimet

Suotimet ovat laajaan kuva-alaan vaikuttavia operaatioita, jotka laskevat jokaiselle pikselille uuden väriarvon vanhojen väriarvojen perusteella. Suotimen määrittelee aina suodinfunktio $f_s(B, i, j, I) \rightarrow c_{ij}$, missä B on suodatettava bittikartta, I suotimen intensiteetti ja i ja j bittikartan koordinaatit. Tuloksena saadaan c_{ij} eli koordinaattien mukaiseen pikseliin sijoitettava uusi väriarvo. Tämän lisäksi suotimilla saattaa olla omia parametrejaan. Yksinkertaisia suotimia ovat esimerkiksi erilaiset värien säädöt sekä kuvan sumennukset ja tarkennukset. Suotimia käytetään laajalti erityisesti valokuvien käsittelyssä, mutta myös muussa graafisessa työssä. Työhistorian kannalta suotimet ovat siksi ongelmallisia, että niiden vaikutusalue on suuri, ja siten suotimen kumoamista varten tarvitsee tallentaa suuri määrä tietoa.

Mikäli halutaan suotimien kohdalla säästää työhistorian vaatimaa muistia suoritusajan kustannuksella, voidaan suotimet tallentaa suodintasoina tasohierarkiaan sen sijaan, että tulokset tallennettaisiin aiemman datan päälle. Tasojen yhdistämisessä laskeaan tällöin suodintason kohdalla pikselin uusi väriarvo suodinfunktion avulla värinsekoitusfunktion sijaan. Tulee huomata, että suodinfunktion parametrina on värinsekoitusfunktiosta poiketen koko alla oleva bittikartta. Tasojen yhdistämisen muistivaatimukset siis kasvavat, kun pikselikohtaista yhdistämisen tulosta ei voida tallentaa suoraan lähdepikselin päälle, ja algoritmi monimutkaistuu.

Kuvassa olevat suodintasot hidastavat kaikkea kuvalle tehtävää käsittelyä jatkossa, mutta työhistoriaan riittää tallentaa vain tieto suodintason luomisesta vaikutusalueen vanhan pikselidatan sijaan. Adobe Photoshop tarjoaa suodintasvoja vaihtoehtona tavallisille suotimille, joskaan kaikkia suotimia ei ole toteutettu tasoina ilmeisesti niiden vaatiman raskaan laskennan takia. Suodintasojen hyötynä on myös käytännöllisempi muokattavuus. Suodintasaan voidaan liittää lisäparametreja, kuten pikselikohtainen intensiteetti-arvo, mikä antaa muistin säästämisen lisäksi myös lisämahdollisuuksia suodinten käyttöön (Fraser & Blatner 2005).

Suodinfunktion laskennallinen monimutkaisuus ratkaisee, mikä toimintatapa on useimmiten järkevä. Useimmat suodinfunktiot, kuten värien säädöt sekä sumennukset voidaan laskea vakioajassa jokaiselle pikselille, kunhan muut parametrit pysyvät vakioina. Vakioajan kerroin saattaa kuitenkin vaihdella suuresti. Esimerkiksi keskiarvotukseen perustuvaa sumennusalgoritmia (Gonzalez & Woods 2008, s. 152-155) käyttävässä sumennuksessa pikseliä kohden vaadittujen kertolaskujen määrä on riippuvainen pikselien määrästä, josta uuden pikselin väriarvo lasketaan.

Sama koskee myös muita konvoluutio-operaatioita, eli operaatioita, joiden tulos määräytyy alkutilasta valittavan suorakulmion muotoisen alueen arvojen lineaarikombinaationa (Gonzales & Woods 2008, s. 146-150). Suodintason käyttämisen jälkeen myös kuvan muokkaus monimutkaistuu: käyttäjä joutuu luomaan uuden tason suodintason päälle pystyäkseen jatkamaan muokkausta samaan tapaan kuin tavallisen suotimen jälkeen jatkaisi. Tämän vuoksi kaikkien suotimien toteuttaminen ainoastaan suodintasoina ei ole käytännöllinen vaihtoehto.

3.8. Kopiointi ja siirtäminen

Suotimien ja siveltimen lisäksi työhistorian kannalta olennaisia toimintoja ovat erilaiset kopioimis- ja siirtämisoperaatiot. Näihin lasketaan valintatyökalun avulla tapahtuva mielivaltaisen muotoisten alueiden pikseleiden siirto toiseen paikkaan, mutta myös muita työkaluja, kuten kloonaustyökalu, transformaatiotyökalu sekä pikseleitä vähän kerrallaan siveltimenomaisesti siirtävä smudge-työkalu.

Yleisesti ottaen kopiointi- tai siirtotyökaluksi määritellään tässä työssä sellainen työkalu, jolla on seuraava siirto-ominaisuus: työkalun käyttämisen jälkeen ainakin yhden pikselin väriarvo koordinaateissa (i_2, j_2) määräytyy osittain sen mukaan, mikä pikselin väriarvo toisissa koordinaateissa $(i_1, j_1) \neq (i_2, j_2)$ oli ennen toiminnon suorittamista. Tämän määritelmän mukaan myös osa suotimista, kuten konvoluutio-operaatiot, täyttävät siirto-ominaisuuden.

Siirto-ominaisuuden täyttävät työkalut eivät vaikuta yhden askeleen kumoamiseen tai kronologiseen kumoamiseen millään tavalla, mutta monimutkaistavat valinnaista kumoamista. Mikäli valinnaisesti kumotun toiminnon jälkeen historiassa on siirto-ominaisuuden täyttävä toiminto, ja toiminnon lähdepikseli koordinaateissa (i_1, j_1) muuttuu valinnaisen kumoamisen seurauksena, myös kohdepikselin koordinaateissa (i_2, j_2) tulisi reagoida tähän, vaikka se ei olisikaan valinnaisesti kumotun toiminnon suoralla vaikutusalueella. Tässä tapauksessa täytyy ottaa käyttöön epäsuoran vaikutusalueen käsite. Toiminnon epäsuoralla vaikutusalueella ovat kaikki suoran vaikutusalueen ulkopuolella olevat pikselit, joiden väriarvo on määräytynyt vaikutusalueen pikseleiden väriarvojen perusteella.

Siirto-ominaisuuden täyttävät työkalut monimutkaistavat joitakin valinnaisen kumoamisen algoritmeja siinä määrin, että saattaa olla mielekäästä hylätä toinen toimintotyyppi osittain tai kokonaan. Yksinkertaisuuden vuoksi valinnainen kumoaminen voidaan sallia ainoastaan silloin, kun siirto-ominaisuuden täyttäviä operaatioita ei ole suoritettu valinnaisesti kumottavan toiminnon jälkeen.

4. TYÖHISTORIAN LISÄSOVELLUKSET BITTIKARTTAEDITORISSA

Kuvan muokkaamiseen liittyvien perustoimintojen lisäksi bittikarttaeditorin työhistoriadata tallennus mahdollistaa myös erilaisia lisätoimintoja. Ensimmäisenä esitellään työhistorian toistaminen animaationa, joka vaatii toimintojen toistamiseen liittyvän datan tallennuksen koko työskentelyn ajalta. Seuraavissa kohdassa käsitellään ylöspäin skaa-lausta työhistorian avulla. Lopuksi käydään läpi muokkausta useammalla samanaikaisella käyttäjällä, mikä asettaa työhistorian toteutukselle aivan omanlaisiaan vaatimuksia.

4.1. Työhistorian toistaminen animaationa

Kuvan muokausprosessi on mahdollista toistaa animaationa työhistoriaan tallennettujen toimintojen toistomahdollisuuden perusteella. Tämän toteuttaminen vaatii, että toistomahdollisuus saadaan tarpeeksi tiiviiseen muotoon, jotta koko työhistorian tallennus pysyvästi toistodatan osalta on mahdollista. Toistodatan lisäksi tarvitaan ainoastaan muutamia pieniä lisäparametreja, kuten kuvan koko ja mahdolliset yleiset ohjelman asetukset, mikäli nämä vaikuttavat työkalujen toimintaan. Mikäli toistaminen halutaan lisäksi tehdä reaaliaikaisesti, työhistoriassa tulisi olla myös toimintojen aikaleimat.

Algoritmisesti toistaminen animaationa on suoraviivainen prosessi. Aluksi alustetaan kuva tallennettujen parametrien perusteella ja ladataan tallennettu työhistoria kokonaisuudessaan kumottujen toimintojen puolelle. Tämän jälkeen pitää vain automaattisesti suorittaa toista-toimintoa kuvalle niin kauan, kunnes työhistoria on käsitelty loppuun. Mikäli halutaan vain toistaa animaatio jatkamatta kuvan muokkausta tästä, voidaan toiminnon toistaminen lisäksi suorittaa ilman, että toimintoja samalla lisätään työhistorian suoritettujen toimintojen puolelle. Tämä säästää suoritusaikaa sekä muistia.

Ainoa monimutkaisempi ongelma työhistorian toistamisessa on reaaliaikaisen toiston tarkkuus. Jotta toistaminen voitaisiin tehdä mahdollisimman uskollisesti alkuperäiselle prosessille, aikaleimojen tulisi olla tarkkoja, ja esimerkiksi siveltimen tapauksessa pitkillä siveltimenvedoilla aikaleimoja pitäisi olla saatavissa myös siveltimenvedon keskeltä. Toiston ajastuksesta sinänsä tulisi myös huolehtia. Lisäksi toisto-ohjelman olisi hyvä havaita katsomista häiritsevät pitkät tauot ja eliminoida nämä. Käyttäjälle olisi myös hyvä antaa mahdollisuus katselun nopeuttamiseen, ja tulisi huolehtia myös siitä, miten toimitaan, jos suorituskyky ei animaation kaikissa kohdissa riitä reaaliaikaiseen toistamiseen.

Hyvänä yleisratkaisuna toisto-ohjelma voi esiprosessoida työhistorian siten, että jokaiselle toiminnolle määritellään olemassa olevien aikaleimojen perusteella tavoit-

teena oleva kesto. Kesto voidaan määritellä nollassa sellaisten työkalujen tapauksessa, joista on saatavilla vain yksi aikaleima. Lisäksi toimintojen väliin sijoitetaan myöskin aikaleimojen perusteella tauko, jolle voidaan asettaa maksimikesto liian pitkien taukojen poistamiseksi. Tällaisen datan perusteella ajastus voidaan hoitaa helposti toiminto kerrallaan ilman kokonaisajan huomioimista, ja yksittäisen toiminnon suoritusajan venyessä yli alkuperäisen ajan seuraava toiminto voidaan suorittaa edelleen normaalisti.

4.2. Ylöspäin skaalaus työhistorian avulla

Mikäli kaikille työkaluille löytyy vektoriesitys, työhistoriaa voi ajatella eräänlaisena hyvin raskaana mutta muutoin tavanomaisena vektorigrafiikkaformaattina. Tämä mahdollistaa erityisesti skaalauksen suurempaan kokoon kuvan sumentumatta tai pikselöitymättä (Eisenberg 2002 s. 1-6). Skaalatessa yksinkertaisesti kerrotaan kaikki ulottuvuuksiin liittyvät parametrit halutulla skaalauskerroimella (Hearn 1986, s. 108-109), ja suoritetaan työhistorian toistaminen ei-reaaliaikaisesti edellisessä kohdassa esiteltyllä menetelmällä. Mikäli työkaluilla ei kuitenkaan ole pätevää vektoriesitystä, lisätarkkuutta ei saavuteta, ja skaalaukseen on parempi käyttää huomattavasti tehokkaampia ja paremmin rinnakkaistettavia signaalinkäsittelyyn pohjautuvia menetelmiä.

Olennaista skaalauksen suhteen on historiadatassa olevien koordinaattien täsmällinen käsittely. Näytöllä olevien pikselien koordinaatit ovat aina kokonaislukuja, mutta mikäli kuvaa suurennetaan esimerkiksi kaksinkertaiseksi, myös koordinaattien tarkkuuden pitäisi kertautua. Ratkaisuna tähän kaikki työhistoriassa olevat toistoon liittyvät koordinaatit on tallennettava desimaalilukuina, samoin kuin kaikki ulottuvuuksiin liittyvät työkalujen parametrit. On myös mielekästä sopia, että pikseleiden kokonaislukukoordinaatit alkuperäisessä kuvassa viittaavat pikseleiden keskipisteisiin. Tällöin esimerkiksi täytettäessä kuvasta neliön muotoinen alue pikseleitä vasemmasta ylänurkasta alueen vasen ylänurkka ei ala koordinaatista (1, 1), vaan koordinaatista (0.5, 0.5).

On jossain määrin kyseenalaista, onko tällaisen ominaisuuden toteuttamisesta työhistorian yhteyteen merkittävää hyötyä loppukäyttäjälle. Kilpailijoita ovat ensinnäkin signaalinkäsittelyn menetelmät, jotka suoriutuvat vastaavasta tehtävästä pikselimäärään verrannollisessa ajassa ja ovat hyvin rinnakkaistettavissa, joskin tuotettuihin kuviin jää havaittavaa sumeutta tai muita suurennuksesta johtuvia artefakteja. Toisaalta haluttaessa parasta mahdollista skaalattavuutta voidaan käyttää puhtaaseen vektorigrafiikkaan paremmin soveltuvia menetelmiä bittikarttaeditorin perinteisten muokkaustyökalujen sijaan.

4.3. Muokkaus useammalla samanaikaisella käyttäjällä

Wang et al. (2002) määrittelevät monen käyttäjän bittikarttaeditorien haasteiksi kaksi ominaisuutta. Ensinnäkin työstettävän dokumentin olisi säilyttävä käyttäjien kesken yhdenmukaisena. Toiseksi käyttäjien tulisi voida kumota ja toistaa omia toimintojaan muiden toimintoihin vaikuttamatta.

Yhdenmukaisuus voidaan varmistaa esimerkiksi asiakas-palvelin-mallisella sovelluksella, jossa palvelin vastaanottaa käyttäjien tuottamat toiminnot, järjestää nämä yksiselitteisesti ja tämän jälkeen lähettää tuloksena syntyneen toimintojonon takaisin käyttäjälle. Käyttäjän sovellus voi tämän jälkeen tutkia, ovatko palvelimelta tulleet toiminnot ristiriidassa käyttäjän paikallisesti suorittamien toimintojen kanssa, ja tämän jälkeen tarvittaessa korjata paikallisen työhistorian palvelimen historiaa vastaavaksi.

Mikäli käytetään asiakas-palvelinmallin sijaan arkkitehtuuria, jossa kaikki käyttäjät ovat tasavertaisessa asemassa keskenään, samanaikaiset toiminnot eri käyttäjiltä tulee järjestää jollakin kehittyneemmällä mekanismilla. Wang et al. (2002) ehdottavat järjestyksen luomiseksi tällaisessa tilanteessa tilavektoriin perustuvaa järjestelmää. Jokaiseen suoritettuun toimintoon liittyvään viestiin liitetään toiminnon alkutilasta laskeutuva tilavektori, jonka perusteella voidaan laskea toiminnon paikka järjestyksessä ja tarvittaessa jäädä odottamaan saapunutta toimintoa ennen suoritettuja toimintoja toisilta käyttäjiltä. Samaan tilavektoriin liittyvät samanaikaiset toiminnot järjestetään käyttäjille luotavan indeksoinnin perusteella.

Jotta käyttäjäkohtainen kumoaminen ja toistaminen onnistuisivat, tieto toimintojen suorittajista on tallennettava ja järjestelmän on tuettava valinnaista kumoamista ja toistamista. Jos nämä ehdot täyttyvät, voidaan käyttäjäkohtainen kumoaminen suorittaa hakemalla viimeisin käyttäjän suorittama toiminto historiasta ja kumota tämä valinnaisella kumoamisella. Tällöin valinnaisen kumoamisen suorituskyky muodostuu todella kriittiseksi sovelluksen toiminnan kannalta, joten työhistorian tulisi olla nimenomaan tätä varten optimoitu.

5. TYÖHISTORIAN TOTEUTUS BITTIKARTTA-EDITORISSA

Tässä luvussa käydään läpi, miten bittikarttaeditorin toimintoja voidaan kumota ja toistaa työn alkupuolella esitetyn työhistoriarakenteen puitteissa. Lähestymistavat etenevät yksinkertaisemmasta monimutkaisempaan: ensin käsitellään pikselidatan tallentamista ja palauttamista, sitten toimintojen toistamista niihin liittyvän vektorimuotoisen syötedatan perusteella ja lopulta johdetaan näistä avainruutuihin perustuva menetelmä työhistorian toteuttamiseen. Lisäksi tarkastellaan Wangin et al. (2002) esittämien tietorakenteiden käyttökelpoisuutta alfakanavan sisältävällä datalla.

5.1. Kuvan tilan tallentaminen

Yksinkertaisin tapa toteuttaa toimintojen kumoaminen ja toistaminen on tallentaa kuvan tila ennen jokaista toimintoa, ja toimintoa kumotessa palauttaa aiempi tila. Kirjaimellisesti toteutettuna tämä tapa vaatii suhteettoman paljon muistia, sillä kaikki kuvaan liittyvä data täytyisi tallentaa jokaisen toiminnon suorittamisen kohdalla. Parempi vaihtoehto on tallentaa toiminnon yhteydessä ainoastaan muutoksen kadottama informaatio, jota varten käytetään jotakin aiemmin esitettyä vaikutusalueen ylärajaa. Vaikutusalueen ylärajan alkutilan pikselidata kopioidaan toiminnon suorittamisen yhteydessä talteen, ja kopioidaan kumoamisen yhteydessä takaisin paikalleen. Toistamisen yhteydessä voidaan vastaavasti kopioida toiminnon tallennettu lopputila takaisin paikalleen.

Tilan tallentamisen perusteella tehtävässä yhden askeleen kumoamisessa tai toistamisessa ei ole laskennallista monimutkaisuutta, joka hankaloittaisi sen suoritusajan arvioimista, sillä operaatiot koostuvat vain tiedon kopioinnista muistipaikasta toiseen. Olettaen, että kopioitavien pikseleiden koordinaatteja ei tarvitse laskea jollakin monimutkaisella menetelmällä, operaation kokonaiskeston määrää suoraan käsitellyn alueen koko. Suoritus aika on siis luokkaa $O(n_p)$, missä n_p on pikseleiden määrä. Ongelmana on ainoastaan suuri muistin käyttö, joka on myöskin luokkaa $O(n_p)$ sekä kumotuille että toistettaville toiminnolle. Varsinkin toimintojen toistamiselle löytyy usein helposti optimaalisempi menetelmä, jollainen esitetäänkin seuraavassa kohdassa.

Tilan tallentamista voi käyttää myös jossain määrin optimoituun kronologiseen kumoamiseen. Kronologinen kumoaminen voidaan suorittaa yhtä bittikarttaa käsitellessä käänteisessä järjestyksessä. Jokaisen toiminnon kumoamisen yhteydessä tarkistetaan, onko toiminnon vaikutusalueen pikselit jo käsitelty, ja jos on, siirrytään suoraan seuraavaan toimintoon tekemättä kuvaan muutoksia. Tämän suorituskyky riippuu erityisesti siitä, kuinka nopeasti vaikutusalueiden ylärajojen leikkaus toiminnolle on lasketta-

vissa. Valinnaiseen kumoamiseen taas tilan tallentaminen ei sovellu lainkaan, sillä jäljempänä historiaan tallennetut tilat ovat riippuvaisia kaikista aikaisemmin suoritetuista toiminnoista (Wang et al. 2002).

5.2. Toistaminen vektoriesityksen perusteella

Toimintojen toistaminen on mahdollista tilan tallentamisen perusteella aivan kuten toimintojen kumoaminenkin. Vaihtoehtoinen ratkaisu on tallentaa sopivasti käsitellyssä muodossa se syötedata, millä toiminto alunperin suoritettiin, ja näin tuottaa eräänlainen vektoriesitys toiminnolle. Tätä on sivuttu jo aiemmin edellisen luvun joidenkin kohtien yhteydessä. Toiminto voidaan sitten toistaa laskemalla sen seuraukset uudelleen aivan kuten työkalua käytettäessä. Syötedatan koko on usein merkittävästi pienempi kuin muuttuneen bittikarttadatan koko, joten tämä on hyvä tapa säästää muistia suorituskyvyn kustannuksella.

Kaikkien toimintojen toisto tällä tavalla ei kuitenkaan ole aivan suoraviivaista. Syötelaitteilta tuleva data on aina mahdollista tallentaa suoraan, mutta joskus tämä ei riitä. Wang et al. (2002) esittävät esimerkkinä ongelmallisesta operaatiosta täyttötyökalussa käytettävän flood fill -algoritmin, jonka kattama alue riippuu alla olevasta bittikartasta. Mikäli halutaan tallentaa algoritmin kuvalle aiheuttama muutos itsessään ilman oletusta siitä, että alla oleva bittikartta pysyy samana, tulee tallentaa täytetty alue pelkän syötelaitteelta tulevan operaation lähtöpisteen sijaan.

Tällaisia vaatimuksia tulee vastaan erityisesti useamman samanaikaisen käyttäjän sovelluksissa, joita Wang et al. (2002) tutkimuksessaan käsittelevät. Wang et al. (2002) päätyvät johtopäätökseen, että työkalujen lopputulos olisi aina syytä tallentaa työhistoriaan raakana pikselidatana, mutta tälle on kuitenkin esitettävissä vastaperusteluja. Esimerkiksi mainitun täyttötyökalun kohdalla voidaan tallentaa täytetyn alueen rajat vektorimuodossa. Tässä tutkimuksessa oletetaan, että kaikille työkaluille on kehitetty vektoriesitys, joiden avulla toiminnot voidaan toistaa.

Jos toistaminen vektoriesityksen perusteella onnistuu, myös valinnainen kumoaminen on mahdollista toteuttaa suoraviivaisesti. Algoritmina historiaan merkitään ensin viimeisin toiminto. Tämän jälkeen kumotaan toiminnot kronologisella kumoamisella aina valinnaisesti kumottuun toimintoon asti, minkä jälkeen siirrytään toistamaan kumotun toiminnon jälkeen tulleet toiminnot vektoriesityksen perusteella. Valinnaisesti kumotun toiminnon toistaminen onnistuu vastaavasti kumoamalla kronologisesti toistettavan toiminnon kohdalle, ja sitten toistamalla toimintoja tästä toiminnosta alkaen. Tällaisella valinnaisella kumoamisella on se etu, että siirto-ominaisuuden täyttävillä työkaluilla ei ole vaikutusta.

5.3. Kumoaminen avainruutujen perusteella

Vektorimuotoinen data on mahdollista tallentaa huomattavasti pienempään tilaan kuin varsinainen pikselidata. Tämän avulla vaadittua muistia voi säästää myös kumoamisen yhteydessä suoritinkäytön ja toteutuksen yksinkertaisuuden kustannuksella.

Avainruutuihin perustuvassa menetelmässä tallennetaan pikselidataa kohdassa 5.1 esitetyn menetelmän tapaan, mutta ainoastaan joidenkin suoritettujen toimintojen kohdalla. Näiksi avaintoiminnoiksi voidaan valita säännöllisesti joka n :s toiminto, tai käyttää toimintojen valitsemiseen jotain monimutkaisempaa valintatapaa. Kutsutaan avaintoiminnon tallennettavaa alkutilaa avainruuduksi. Kun tämän jälkeen suoritetaan uusia toimintoja ennen seuraavaa avaintoimintoa, tallennetaan nämä ainoastaan toiston mahdollistavassa vektorimuodossa. Lisäksi täydennetään edellistä avainruutua lisäämällä siihen niiden alueiden pikselidata, jotka kuuluvat suoritettavan toiminnon vaikutusalueeseen, mutta eivät kuulu alkuperäiseen avainruutuun.

Nyt kronologinen kumoaminen voidaan suorittaa seuraavasti. Palautetaan ensin aikaisinta kumottua toimintoa edeltävän avaintoiminnon alkutila käyttämällä tilan tallentamisen kronologisen kumoamisen algoritmia avainruuduille. Lisäksi siirretään kumottavat toiminnot suoritetuista toiminnoista kumottuihin. Tämän jälkeen käydään läpi palautetusta avaintoiminnosta alkaen suoritettuja toimintoja järjestyksessä ja suoritetaan näiden toisto-operaatiot, kunnes saavutetaan aikaisin kumottu toiminto. Yhden askeleen kumoamisoperaatio on johdettavissa tämän erikoistapauksena. Olennaiseksi kustannukseksi muodostuu valitun avaintoiminnon jälkeen suoritettujen toimintojen toistamisen vaatima laskenta, ja tämän perusteella pitäisikin ratkaista, kuinka usein avainruutuja tallennetaan. Mikäli jokainen toiminto valitaan avaintoiminnoksi, algoritmin suoritus on identtinen tilan tallentamisen kanssa.

Valinnaisen kumoamisen algoritmi toimii samalla periaatteella kuin edellisessä kohdassa. Käytetään ensin kronologista kumoamista palauttamaan valinnaisesti kumottua toimintoa edeltävä tila, ja tämän jälkeen toistetaan kaikki menneet toiminnot tästä alkaen lukuun ottamatta kumottua toimintoa.

Algoritmin hyöty muihin vaihtoehtoihin nähden riippuu paljon siitä, miten käyttäjä käyttää bittikarttaeditoria. Mikäli peräkkäiset toiminnot kohdistuvat kaikki samalle bittikartan alueelle, tilaa voidaan säästää merkittävästi. Tarkkaa arviota tilansäästön ja vaaditun lisälaskennan vaihteluväleistä on kuitenkin mahdoton tehdä ilman laajaa tilastollista dataa bittikarttaeditorien käyttötavoista. Avainruutumenetelmän parametrien optimointi on sekin itsessään monimutkainen ongelma.

5.4. Päällekkäisyysgraafi

Edellä esitetyt menetelmät toteuttavat ainakin yhden askeleen kumoamisen ja myös kronologisen kumoamisen vähintäänkin tyydyttävästi. Valinnainen kumoaminen vaatii kuitenkin suhteellisen paljon suoritusaikaa, mikä muodostuu ongelmaksi varsinkin usean käyttäjän sovelluksissa, joissa valinnainen kumoaminen on yleinen operaatio.

Useamman käyttäjän sovellukset ovat siten vaikuttaneet muiden, monimutkaisempien historiadatan tallennusmenetelmien kehittämiseen. Erään tällaisen menetelmän, quadtree-pohjaisen toimintograafin (QODG, Quadtree-based Operation Distribution Graph), ovat kehittäneet Wang et al. (2002). Tässä menetelmässä toiminnoista muodostetaan päällekkäisyysgraafi (DGS, Directed Graph Structure), jossa toiminnot on kytketty toisiinsa sen perusteella, vaikuttavatko ne samoihin kuvan alueisiin. Lisäksi käytetään Quadtree-tietorakennetta (QODG), jossa kuva on jaettu rekursiivisesti samankokoisiin neliöihin niin, että lopulta kuhunkin neliöön vaikuttaa vain yksi toiminto. Tämän avulla voidaan nopeuttaa DGS:lle tehtäviä operaatioita.

Tulee kuitenkin huomata, että menetelmä on kehitetty työkaluille, jotka eivät tuota alfa-kanavaa sisältävää dataa. Mikäli toiminnot ovat osittain läpinäkyviä, käykin niin, että yhteen pikseliin voi vaikuttaa useampi toiminto, eikä QODG-jakoa voida tehdä yhteen toimintoon per neliö. Tämä rikkoo QODG-rakenteen perusajatuksen, joten Wangin et al. (2002) menetelmä ei ole suoraan sovellettavissa tämän työn esimerkkisovellukseen. Wangin et al. menetelmässä oletetaan myös DGS:ää päivitettäessä, että toiminnot peittävät vaikutusalueellaan kokonaan allaolevat toiminnot.

Sovellettaessa Wangin et al. menetelmää alfakanavan sisältävälle datalle olisikin mielekästä ottaa käyttöön ainoastaan sovellettu DGS-rakenne, jonka avulla voidaan päätellä, mitkä toiminnot on mahdollisesti laskettava uudelleen. Käytetään DGS:n toiminnoista merkintää O , ja otetaan Wangin et al. tapaan käyttöön käsite toiminnon osoitinlista ($O.PL$), jossa on toimintoa kumottaessa toiminnon alta näkyviin ilmestyvät toiminnot. Wangin et al. menetelmästä poiketen merkitään kuitenkin osoitinlistaan enemmän toimintoja: listaan tulevat kaikki sellaiset toiminnot, jotka on suoritettu toimintoa ennen ja ovat toiminnon suoran vaikutusalueen alla.

Lisäksi määritellään uutena ominaisuutena seuraajalista ($O.SL$) jossa on kaikki ne toiminnot, joilla on O osoitinlistassaan. Molempien listojen osoittamat toiminnot on järjestettävä niiden kronologisen järjestyksen perusteella. Toiminnon vaikutusalueesta käytetään tässä sovelletussa ratkaisussa merkintää $O.alue$. Toiminnolla on lisäksi piirrä-funktio, joka suorittaa toiminnon ja ottaa parametrinaan pikselijoukon, minkä alueelle lopputuloksen piirto-operaatiot rajoitetaan.

Menetelmän käyttö monimutkaistaa uusien toimintojen suorittamista, sillä ne on lisättävä myös DGS-rakenteeseen, jolloin niille on luotava osoitinlista ja päivitettävä vanhojen toimintojen seuraajalistaa vastaavasti. Uuden toiminnon osoitinlistaan lisättäviä toimintoja voidaan alustavasti karsia toimintojen määrään verrannollisessa ajassa käyttämällä toimintojen vaikutusalueille suorakulmion muotoista ylärajaa. Kun tämä karsinta on tehty, tarkempi karsinta jäljellejäävistä toiminnoista täytyy suorittaa laske-malla näiden vaikutusalueen leikkaus uuden toiminnon vaikutusalueen kanssa.

```

function piirrä_seuraajat (O, PA)
    if (PA  $\cap$  O.alue = NULL)
        return false
    end if
    piirrä (O, PA)
    foreach Oq in O.SL:
        piirrä_seuraajat (Oq, PA  $\cap$  O.alue)
    end foreach
    return true
end function

function piirrä_edeltäjät(O, PA)
    if (PA  $\cap$  O.alue = NULL)
        return false
    end if
    foreach Op in O.PL:
        piirrä_edeltäjät(Op, PA  $\cap$  O.alue)
    end foreach
    piirrä(O, PA)
    return true
end function

function kumoa(Ok)
    for Op in Ok.PL:
        piirrä_edeltäjät (Op, Ok.alue)
        for Oq in Ok.SL:
            if (piirrä_seuraajat (Oq, Op.alue  $\cap$  Ok.alue))
                Oq.PL.lisää(Op)
                Op.SL.lisää(Oq)
            end if
        end for
    end for
end function

```

Kuva 5.1: DGS-pohjainen valinnainen kumoaminen alfakanavan sisältävälle pikselidatalle, yleistetty Wangin et al. (2002) antaman algoritmin pohjalta.

DGS-pohjainen valinnainen kumoaminen onnistuu algoritmilla, joka on annettu kuvassa 5.1. Jotta annettua algoritmia voisi soveltaa mielekkäästi, toiminnoista on oltava nopeasti saatavilla kaikkien vaikutusalueen pikseleiden koordinaatit sekä toiminnon näille aiheuttamat värinmuutokset. Mielekkäin vaihtoehto on tallentaa nämä muistiin, joten algoritmin vaatima muisti on jotakuinkin samassa suuruusluokassa optimoidun tilan tallentamisen kanssa. Kuitenkin suoritusajan kustannuksella voidaan tätä menetelmää käyttäen tallentaa toiminnoista ainoastaan vektorimuotoinen data, ja säästää muistia merkittävästi.

Ongelmaksi muodostuu algoritmin suoritus aika silloin, kun toimintoja on historiassa paljon: pahimmillaan rekursiivinen piirrä_seuraajat-funktio tai piirrä_edeltä-

jät-funktio suoritetaan kumoamisen yhteydessä jokaiselle historian toiminnolle useita kertoja. Vaikutusalueiden leikkausten laskeminen vie tällöin erityisen paljon aikaa. Mikäli vaikutusalueet ovat saatavilla pikselikoordinaattien järjestettyinä listoina koillaan n_p ja m_p , leikkauksen laskeminen on $O(n_p + m_p)$ operaatio – molemmat listat käydään läpi rinnakkain järjestyksessä. Mikäli vaikutusalueet tallennetaan suorakulmion muotoisina bittikarttoina, joista pienemmässä on n_p pikseliä, leikkauksen laskeminen on $O(n_p)$ operaatio – pikselit voidaan käydä läpi lineaarisesti molemmista bittikartoista pienemmän bittikartan koordinaattien perusteella.

Paras väline suoritusajan rajoittamiseen DGS-pohjaisen algoritmin yhteydessä on melko aggressiivinen työhistorian muistinhallinta. Tavanomaisten parametrien lisäksi muistinhallinta voi tarkkailla operaatioiden osoitinlistojen ja operaatioiden vaikutusalueiden keskimääräistä kokoa sekä myös sitä, kuka käyttäjä on suorittanut minkäkin toiminnon, ja leikata kumottavissa olevia operaatioita näiden perusteella. Jotta leikkaus olisi ylipäättään mahdollista, työhistorian pohjalle DGS:ään on tarpeen lisätä ylimääräinen operaatio, joka alustaa kuvan työhistoriasta leikattujen operaatioiden tuottamalla kuvalla. Kuvaa voidaan päivittää aina leikatessa työhistoriasta vanhoja operaatioita, ja sen avulla työhistorian leikkaaminen mahdollistuu ilman, että varsinaiseen kumoamisalgoritmiin tarvitaan lainkaan muutoksia.

DGS-pohjaisen algoritmin yhteydessä on huomattava myös toimintojen siirto-ominaisuuden vaikutus. Toimintojen epäsuora vaikutusalue tulee ottaa DGS:n osoitinlistojen rakentamisessa huomioon, ja epäsuorat vaikutukset tulee erikseen laskea valinnaisen kumoamisen yhteydessä, mikäli siirto-ominaisuuden täyttävät työkalut sallitaan. Tämä monimutkaistaa algoritmin käytännön toteutusta entisestään.

Tällaisenaan DGS-pohjainen algoritmi ei ole kovin suoraviivaisesti toteutettavissa, ja sen analyysi on hankalaa. Jatkotutkimusta tarvitaan, ennen kuin menetelmää voidaan soveltaa käytäntöön.

6. JOHTOPÄÄTÖKSET

Työssä tuli esille työhistorian toteutuksen suhteen useita erilaisia kriteereitä, jotka ovat osittain keskenään ristiriidassa. Työhistorian tallennusmenetelmä tulisikin lopulta valita sen mukaan, millaisia ominaisuuksia sovellukselta halutaan. Lisäksi tallennusmenetelmän ja algoritmien valintaa ja parametrien optimointia varten tulisi suorittaa lisää käytännön kokeita, joissa selvitetään historian ja työkalujen todelliset käyttötavat ja näiden vaikutus käyttökokemukseen. Tilastolliselle tiedolle aiheesta olisi paljon käyttöä arvioitaessa erilaisten työhistoria-algoritmien soveltuvuutta käytäntöön.

Mikäli valinnaista kumoamista ei haluta toteuttaa lainkaan tai sitä käsitellään lisäominaisuuden luonteisesti, yksinkertainen tilan tallentaminen on varmin työhistoria-algoritmi. Sen suoritus aika sekä tilavaatimukset ovat helposti laskettavissa ja yhden askeleen kumoamiselle hyvin kohtuullisia, se ei ole ristiriidassa siirto-ominaisuuden kanssa ja myös sen toteutus on yksinkertaista. Tilan tallentamista voidaan myös helposti täydentää koko työskentelyhistorian ajalta tallennettavalla toistohistorialla, joka mahdollistaa työhistorian toiston jälkikäteen animaationa.

Tilan tallentamisen yhteydessä tarvittavaksi toiminnon vaikutusalueen ylärajaksi voidaan suositella vaikutusalueen jaottelua erikokoisiin neliöihin. Tämä tarjoaa optimointimahdollisuuksia vaikutusalueiden erotuksen laskemiseen kronologisen kumoamisen yhteydessä, ja kuluttaa suhteessa vähän tilaa tallennettavaan pikselimäärään nähden. Dynaaminen neliöihin jaottelu kuluttaa kuitenkin suoritus aikaa, ja epäoptimaalisestakin jaottelualgoritmista saattaa muodostua monimutkainen, joten kiinteän ruudukkorakenteen pohjalta jaottelu on sekin suositeltavissa.

Monen käyttäjän sovelluksissa pääpaino on kuitenkin valinnaisella kumoamisella, ja tähän yksinkertainen tilan tallentaminen soveltuu huonosti. Mikäli luovutaan työkalujen läpinäkyvyydestä, voidaan käyttää Wangin et al. (2002) algoritmia suoraan – tämä olisi kuitenkin nykyaikaisessa editorissa kohtuuton vaatimus. Järkeviksi vaihtoehtoiksi jäävät avainruutujen käyttö tai sovellettu DGS-pohjainen järjestelmä, jotka kummatkin ovat kuitenkin suoritusajan suhteen melko raskaita ja vaativat tarkempaa jatkotutkimusta ennen käytännön sovellettavuutta. Monimutkaisuutta lisäävät DGS-pohjaisen järjestelmän kohdalla myös siirto-ominaisuuden täyttävät työkalut.

Onkin ymmärrettävää, että joissakin nykyisissä monen käyttäjän taiteilijoille suunnatuissa bittikarttaeditoreissa toimintojen kumoamista ei ole mahdollistettu lainkaan. Vaihtoehtona tälle voitaisiin kuitenkin tutkia myös käyttäjien kesken jaetun kumoamisen käytettävyyttä, tai mahdollisuuksia erotella eri käyttäjien käsittelemä data editorissa siten, että jokaisella käyttäjällä voisi olla oma, erillinen työhistoriansa. Tällöin monen käyttäjän sovellusten tavalliset ongelmat poistuisivat, ja työhistoriaoperaatioihin voitaisiin käyttää yksinkertaisimpia menetelmiä.

LÄHTEET

Adobe Systems, 2008: image editor | Adobe Photoshop CS4 [viitattu 19.1.2009]. Saatavissa:

<http://www.adobe.com/products/photoshop/photoshop/>

Eisenberg, J. David: SVG Essentials, O'Reilly, Sebastopol, CA, USA, 2002. 335 sivua.

Fraser, Bruce & Blatner, David: Photoshop tutorial: Master advanced adjustment layer techniques, katkelma kirjasta Real World Adobe Photoshop, Peachpit Press, 2005. [viitattu 3.2.2009]. Saatavissa:

http://www.adobe.com/designcenter/photoshop/articles/phscs2it_adjustlayer.html

Gonzalez, Rafael & Woods, Richard: Digital Image Processing: Third Edition, Prentice Hall, Upper Saddle River, New Jersey, 2008. 954 sivua.

Hearn, Donald: Computer Graphics, London: Prentice-Hall International, 1986. 352 sivua.

Hsu, Siu Chi & Lee, Irene H. H.: Drawing and animation using skeletal strokes, Proceedings of the 21st annual conference on Computer graphics and interactive techniques, ACM New York, NY, USA, 1994. 10 sivua.

Kimball, Spencer & Mattis, Peter: 1995, gimpimage-merge.c GIMP versio 2.6.4. [viitattu 26.1.2009]. Saatavissa:

<ftp://ftp.gimp.org/pub/gimp/v2.6/gimp-2.6.4.tar.bz2> , <app/core/gimpimage-merge.c>

Sun, C.: Undo any operation at any time in group editors. Proceedings of ACM Conference on Computer Supported Cooperative Work. ACM, Philadelphia, PA, USA, 2000. 10 sivua.

The GIMP Team, 2009: GIMP – The GNU Image Manipulation Program [viitattu 19.1.2009]. Saatavissa:

<http://www.gimp.org/>

Wang, Xueyi, Bu, Jiajun & Chen, Chun: Achieving undo in bitmap-based collaborative graphics editing systems. Proceedings of the 2002 ACM conference on Computer supported cooperative work, ACM New York, NY, USA, 2002, 9 sivua.

Weisstein, Eric: Quadtree. From MathWorld - A Wolfram Web Resource. [viitattu 7.3.2009]. Saatavissa:
<http://mathworld.wolfram.com/Quadtree.html>

Whitted, Turner: Anti-aliased line drawing using brush extrusion. ACM SIGGRAPH Computer Graphics volume 17, issue 3, ACM New York, NY, USA, 1983, 6 sivua.

Wu, Xiaolin: An Efficient Antialiasing Technique. Proceedings of the 18th annual conference on Computer graphics and interactive techniques. ACM New York, NY, USA, 1991, 10 sivua.

LIITE 1: JAVA-ALGORITMI DYNAAMISEN NELIÖIHIN JAKAMISEN ANALYSOINTIIN

```

public class Layer {

    protected int width;
    protected int height;
    protected int[] bitmap;

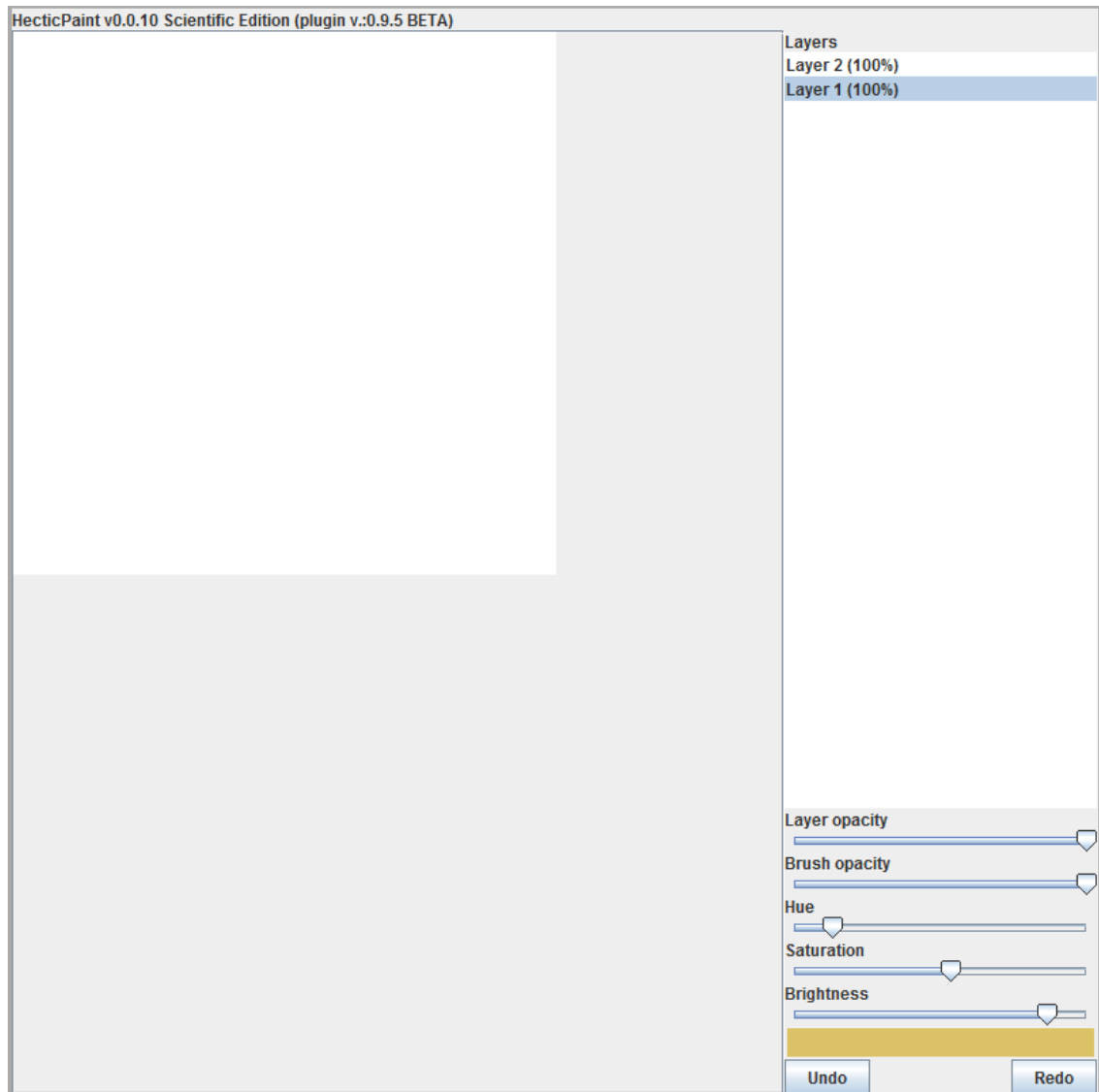
    ...

    /* boxWidth : laatikkokoko, jolle jako tällä iteraatiolla tehdään.
       minBoxWidth : laatikkokoko, mihin asti iteroidaan.
       aB : taulukko, joka sisältää iterointialueen rajat. Reunat ovat mukana alueessa. */
    private Object[] analyzeQuad(int boxWidth, int minBoxWidth, int space, int[] aB)
    {
        int filledPixels = 0; // Täytetyt pikselit aB:n sisällä yhteensä
        int savedPixels = 0; // Ylärajan sisälle jääneet ts. tallennettavat pikselit
        for (int xbox = 0; xbox < (aB[2] - aB[0]) / boxWidth + 1; ++xbox)
        {
            for (int ybox = 0; ybox < (aB[3] - aB[1]) / boxWidth + 1; ++ybox)
            {
                int boxExtraPixels = 0;
                int boxFilledPixels = 0;
                boolean boxInUse = false;
                for (int x = aB[0] + xbox * boxWidth; x < aB[0] + (xbox + 1) * boxWidth; ++x)
                {
                    for (int y = aB[1] + ybox * boxWidth; y < aB[1] + (ybox + 1) * boxWidth; ++y)
                    {
                        if (x > aB[2] || y > aB[3])
                        {
                            ++boxExtraPixels; // Alue menee yli ison laatikon tallennetun rajojen,
                                                // näitä pikseleitä ei tarvitse ottaa laskennassa huo-
                                                // mioon.
                        }
                        else if ((bitmap[y * width + x] & 0xFF000000) != 0)
                        {
                            // Tässä käytetään muusta työstä poiketen ARGB-formaattia.
                            // Ehtolauseke kertoo siis, onko pikseli ei-läpinäkyvä.
                            ++boxFilledPixels;
                            boxInUse = true;
                        }
                    }
                }
            }
        }
        if (boxInUse)
        {
            if (boxFilledPixels > (boxWidth * boxWidth - boxExtraPixels) - 5 * 4
                || boxWidth / 2 < minBoxWidth)
            {
                // Neliötä käytetään suoraan, kun se on optimaalisin vaihtoehto
                filledPixels += boxFilledPixels;
                savedPixels += boxWidth * boxWidth - boxExtraPixels;
                space += (boxWidth * boxWidth - boxExtraPixels) * 4 + 5;
            }
            else {
                // Rekursiivinen jako pienempiin neliöihin tehdään, jos näin saatetaan
                // säästää tilaa
                int[] newBounds = {aB[0] + xbox * boxWidth,

```

```
        aB[1] + ybox * boxWidth,
        Math.min(aB[0] + (xbox + 1) * boxWidth - 1,
        aB[2]), Math.min(aB[1] + (ybox + 1) * boxWidth - 1, aB[3]));
    Object[] recursive = analyzeQuad(boxWidth / 2, minBoxWidth, 0, newBounds);
    savedPixels += (Integer)(recursive[0]);
    space += (Integer)(recursive[1]);
    filledPixels += (Integer)(recursive[2]);
}
}
}
}
return new Object[]{new Integer(savedPixels),
                    new Integer(space), new Integer(filledPixels)};
}
...
}
```

LIITE 2: ANALYYSISSÄ KÄYTETYN BITTIKARTTA-EDITORIN KÄYTTÖLIITTYMÄ



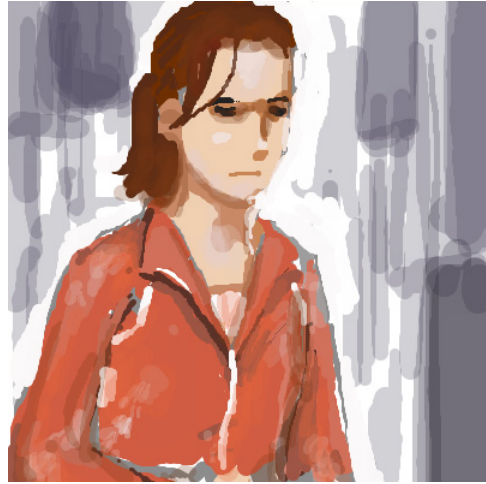
Kuva: Bittikarttaeditorin käyttöliittymä. Käyttöliittymässä säätimet oikealla ylhäältä alaspäin: Tason valinta listasta, tason läpinäkyvyys, siveltimen läpinäkyvyys, värin sävy, värin kylläisyys, värin kirkkaus, värin esikatselu, kumoa, toista.

Käytetty editori, HecticPaint, on aikaisessa kehitysvaiheessa oleva Java-bittikarttaeditori, jota on kehitetty osittain tämän työn tarkoituksia varten.

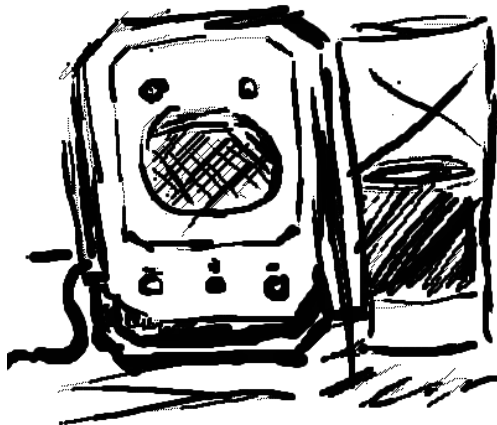
LIITE 3: KOEPIIRTÄJIEN PIIRROKSET



1.



2.



3.



4.

Kuva: 1. Koepiirtäjä A:n viivapiirros, 2. koepiirtäjä A:n väritetty piirros, 3. Koepiirtäjä B:n viivapiirros, 4. Koepiirtäjä B:n väritetty piirros.

LIITE 4: KOEPIIRTÄJIEN TEHTÄVÄNANTO

Lue ohjeet kokonaan läpi, ennen kuin teet mitään.

1. Kokeile ensin sovellusta varmistaaksesi, että ymmärrät mitä säätimet tekevät. Avaa lisäksi Java-konsoli ja varmista, että sinne ilmestyy numeroita aina kun piirrät jotain. Java-konsolin saa auki Javan tehtäväpalkin kuvakkeesta, joka ilmestyy näkyviin kun Java-appletti käynnistyy. Konsolin saat auki klikkaamalla kuvaketta hiiren oikealla painikkeella ja valitsemalla sieltä Open [versionumerosi] Console.

Mikäli kuvaketta ei näy, mene ohjauspaneeliin ja valitse Java. Merkitse Advanced-ta-bilta ominaisuus Miscellaneous -> Place Java icon in system tray. Mikäli Javaa tai tätä asetusta ei löydy Ohjauspaneelistä, tai Java-sovelma ei peräti käynnisty ollenkaan, lataa [Sunin Java-virtuaalikoneen uusin versio](#) ja asenna se sekä tämän jälkeen *JTablet*. Mikäli JTablet on varmasti asennettuna, mutta paineentunnistus ei toimi, lataa sivu uudelleen.

2. Lataa sivu uudelleen aloittaaksesi puhtaalta pohjalta. Piirrä viivapiirros vapaasta aiheesta vapaalla tyylillä. Käytä aikaa noin 5 minuuttia. Ota tämän jälkeen kaikki Java-konsolin tulosteet talteen analysointia varten ja ruutukaappaus piirtämästäsi kuvasta. Laita kuvan tulosteet omaan tekstitiedostoonsa, jonka nimeät samannimiseksi kuin kuvatiedoston.

3. Lataa sivu uudelleen aloittaaksesi puhtaalta pohjalta. Piirrä väritetty kuva vapaasta aiheesta vapaalla tyylillä. Käytä aikaa noin 5 minuuttia. Ota tämän jälkeen kaikki Java-konsolin tulosteet talteen analysointia varten ja ruutukaappaus piirtämästäsi kuvasta.